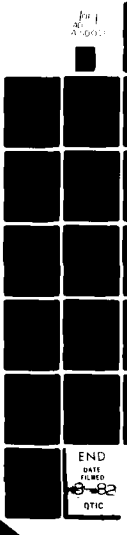
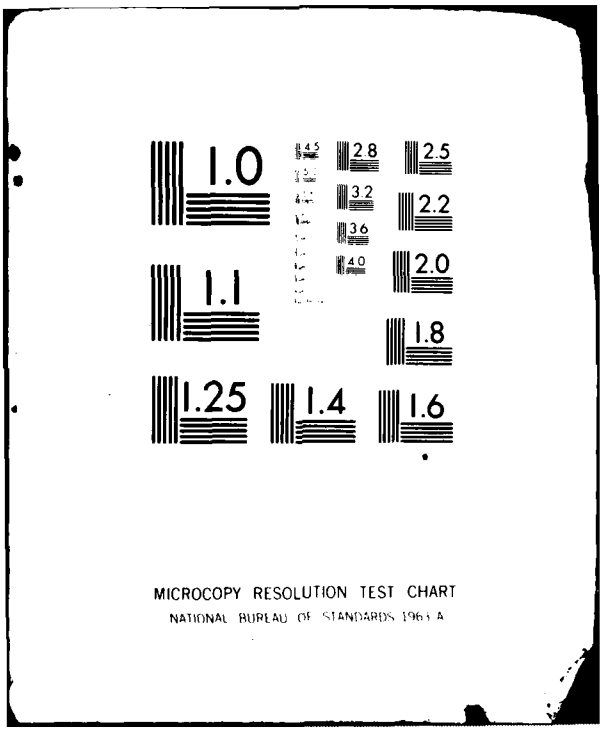
AD-A110 001 INTERMETRICS INC CAMBRIDGE MA F/G 9/2
ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECI--ETC(U)
DEC 81 F30602-80-C-0291
UNCLASSIFIED RADC-TR-81-358 NL



END
DATE FILMED
8-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

PHOTOGRAPH THIS SHEET

AD A110001

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Intermetrics, Inc
Cambridge, MA
ADA Integrated Environment I Computer Program Development Specification. Interim Rept 15Sep.80-15Mar81 Dec.81
DOCUMENT IDENTIFICATION

Contract F30602-80-C-0291 RADC-TR-81-356, Vol.III

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
S JAN 25 1982 D
D

DATE ACCESSIONED

COPY INSPECTED

82 01 12 016

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DOCUMENT PROCESSING SHEET

DTIC FORM OCT 79 70A

RADC-TR-81-358, Vol III (of seven)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-358, Volume III (of seven) has been reviewed and is approved for publication.

APPROVED:

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-81-358, Vol III (of seven)	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION	5. TYPE OF REPORT & PERIOD COVERED: Interim Report 15 Sep 80 - 15 Mar 81
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s): F30602-80-C-0291
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Intermetrics, Inc. 733 Concord Avenue Cambridge MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62204F/33126F 55811908
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COES) Griffiss AFB NY 13441	12. REPORT DATE: December 1981
	13. NUMBER OF PAGES: 75
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE OF CONTENTS

		PAGE
1.0	SCOPE	1
1.1	Identification	1
1.2	Functional Summary	1
2.0	APPLICABLE DOCUMENTS	3
2.1	Government Documents	3
2.2	Non-Government Documents	3
3.0	REQUIREMENTS	5
3.1	Program Definition	5
3.1.1	Interface Requirements	5
3.1.1.1	The CP/KAPSE Interface	5
3.1.1.2	CP/Fundamental Programs	5
3.1.1.3	CP/User Programs	7
3.1.2	Main Constituent Compilation Units	9
3.2	Detailed Functional Requirements	11
3.2.1	The MCL Command Language	11
3.2.1.1	User Commands	11
3.2.1.2	Language Elements	35
3.2.2	Processing	44
3.2.2.1	The CP Driver (DRIVER)	44
3.2.2.2	Input Stream Package (INPUT_STREAM)	47
3.2.2.3	Lexical Analyzer	47
3.2.2.4	The MCL Parser (PARSE)	48
3.2.2.5	The Parse-Tree Interpreter (TREE INTERPRET)	48
3.2.2.6	The Globals Package (GLOBALS)	51
3.2.2.7	The Interpreter Allocation Task (ALLOCATE INTERPRETER)	51
3.2.2.8	The Background Task Manager (BACKGROUND)	52
3.2.2.9	Job Invocation (JOB)	53
3.2.2.10	Subprogram Simulation Package (SUBPROG_SIMULATE)	54
3.2.2.11	Variable Task (VARIABLE)	55
3.2.2.12	String Conversion Package (STRING_CONVERT)	56

TABLE OF CONTENTS (Cont'd.)

	PAGE
3.2.2.13 Operator Package (OPERATOR)	57
3.2.2.14 Expression Package (EXPRESSION)	57
3.2.2.15 Error Package (ERROR)	58
4.0 QUALITY ASSURANCE PROVISIONS	59
4.1 Introduction	59
4.1.1 Unit Testing	59
4.1.2 Integration Testing	59
4.1.3 Functional Testing	59
10.0 APPENDIX	61
10.1 BNF for MCL (MAPSE Command Language)	61
10.2 Statements and Fundamental Programs	71
10.3 Ada - MCL Comparison	73

LIST OF FIGURES AND ILLUSTRATIONS

FIGURE 3-1:	KAPSE FUNCTIONAL RELATIONSHIPS	6
FIGURE 3-2:	PARAMETER PASSING	8
FIGURE 3-3:	CP MODULES	10
FIGURE 3-4:	SAMPLE CP SESSION	14
TABLE 3-1:	COMMAND SUMMARY	12
TABLE 3-2:	JOB ATTRIBUTES	29
TABLE 3-3:	PRE-DEFINED CP VARIABLES	40
TABLE 3-4:	MCL OPERATORS	42
TABLE 3-5:	IMPLICIT TYPE CONVERSIONS	43
TABLE 10-1:	STATEMENTS	71
TABLE 10-2:	FUNDAMENTAL PROGRAMS	72
TABLE 10-3:	COMMANDS	73
TABLE 10-4:	LANGUAGE ELEMENTS	75

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

1.0 SCOPE

1.1 Identification

This specification describes the command language with which a user selects MAPSE facilities, and establishes the requirements for performance, design, test and qualification of the Command Processor(CP), a computer program that interprets and acts upon user commands. This specification also identifies interfaces with the KAPSE and with other MAPSE tools that, together, provide the full range of capabilities available to the MAPSE user.

1.2 Functional Summary

The user communicates with the CP via an Ada-like interpretive language called MCL (MAPSE Command Language). In response to MCL commands, the CP invokes arbitrary programs, as well as performing other related actions.

More specifically, the CP provides the following basic capabilities:

1. invocation of arbitrary tools and user-defined programs, and control over their execution;
2. help on a per-program or per-program-parameter basis;
3. manipulation of CP variables;
4. redirection of standard input and output for arbitrary commands;
5. the ability to connect arbitrary commands as co-routines via pipes;
6. the ability to execute arbitrary commands in the background, and to control their execution; and
7. the processing of scripts of MCL commands, which may be parameterized.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

PAGE LEFT BLANK INTENTIONALLY

2.0 APPLICABLE DOCUMENTS

Please note that the bracketed number preceding the document identification is used for reference purposes within the text.

2.1 Government Documents

[G-1] Statement of Work for Ada Integrated Environment, PR No. B-0-3233, December 1979.

[G-2] Requirements for Ada Programming Support Environment, "STONEMAN", Department of Defense, February 1980.

[G-3] Reference Manual for the Ada Programming Language, proposed standard document, U.S. Department of Defense, July 1980 (reprinted November 1980).

2.2 Non-Government Documents.

[N-1] Diana Reference Manual (G.Goos + Wm. A. Wulf, eds.) Institut Fuer Informatik II, Universitaet Karlsruhe and Computer Science Department Carnegie-Mellon University, March 1981.

[I-1] System Specification for Add Integrated Environment, Type A, Intermetrics, Inc., March 1981, IR-676.

Computer Program Development Specifications for Ada Integrated Environment (Type 5):

[I-2] Ada Compiler Phases, IR-677

[I-3] KAPSE/Database, IR-678

[I-4] MAPSE Generation and Support, IR-680

[I-5] Program Integration Facilities, IR-681

[I-6] MAPSE Debugging Facilities, IR-682

[I-7] MAPSE Text Editor, IR-683

[I-8] Technical Report (Interim) IR-684

PAGE LEFT BLANK INTENTIONALLY

3.0 REQUIREMENTS

3.1 Program Definition

3.1.1 Interface Requirements

The Command Interpreter interfaces with:

1. the KAPSE;

2. fundamental programs;

3. user programs and scripts.

These interfaces are diagrammed in Figure 3-1, and are described below.

3.1.1.1 The CP/KAPSE Interface

When a user logs in to the MAPSE system, the system initiator within the KAPSE invokes the CP on the user's behalf. At the conclusion of the CP session, control is returned to the system initiator.

The CP receives its input in interactive mode from the teletype. The teletype driver within the KAPSE provides the user with primitive editing capabilities such as character deletion.

In carrying out user-specified commands, the CP invokes various KAPSE functions. These are described in detail in Section 3.2.2.

3.1.1.2 CP/Fundamental Programs

Some user commands require manipulation of data structures defined within the CP (e.g., CP variables). These commands are interpreted within the CP. To change the syntax or effect of such commands would require the recompilation of the CP.

Commands that do not reference any internal CP data structures are implemented as linked executable programs called fundamental programs. These are invoked by the CP to carry out specific actions. The syntax or effect of a command implemented via a fundamental program can be modified by recompiling the fundamental program.

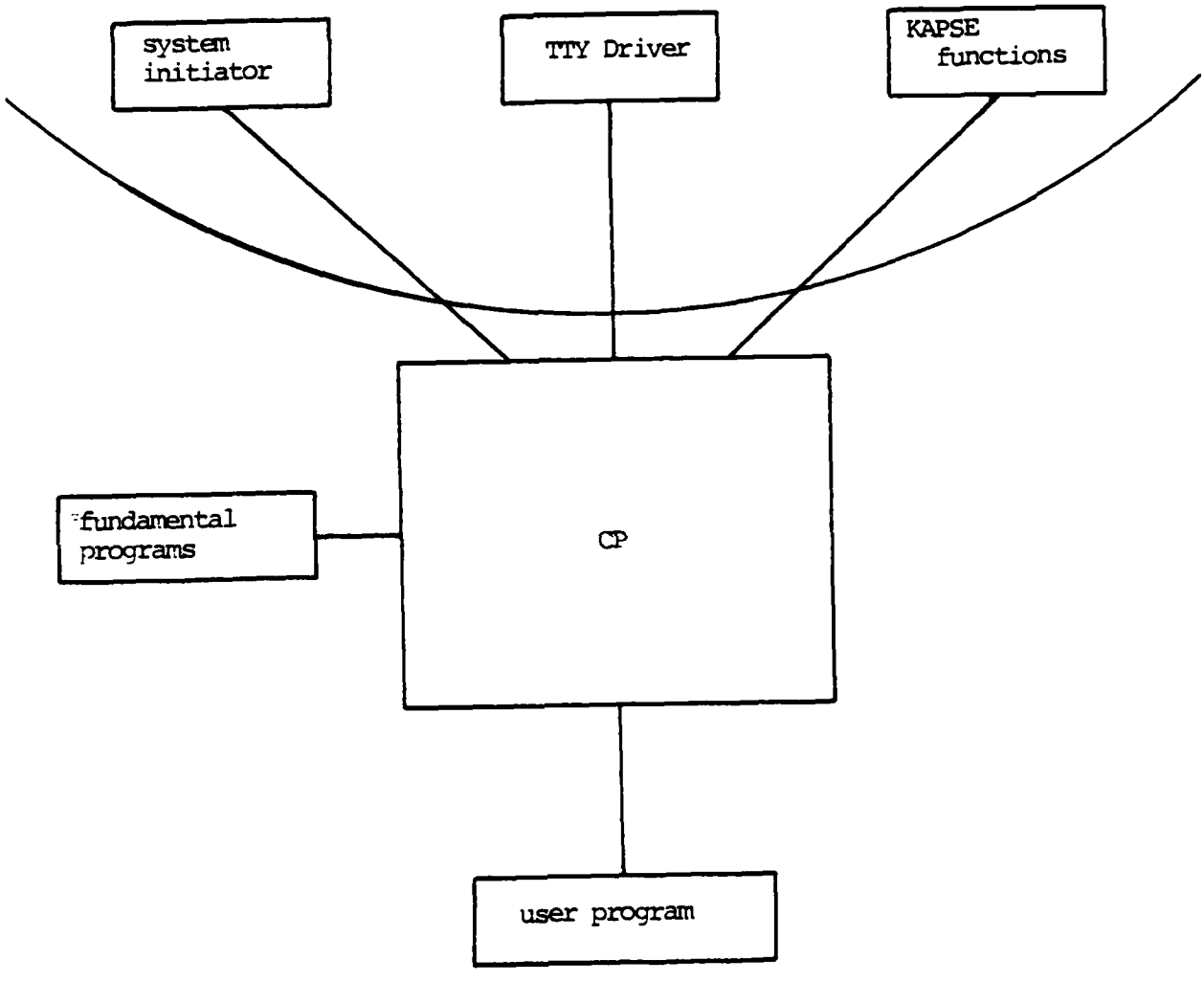


Figure 3-1: KAPSE FUNCTIONAL RELATIONSHIP

The MCL language description uses the term "statement" to indicate those commands that are carried out within the CP. Appendix II lists fundamental programs referenced in this document.

3.1.1.3 CP/User Programs

The CP user can invoke an arbitrary program. As described in Section 3.2.1.1, a program may optionally be supplied with parameters. If the program does not require parameters, the CP simply creates a context object for the program and makes the KAPSE call INITIATE_PROGRAM. If, however, the program does take parameters, special processing is required on the part of the CP and of the invoked program. Parameters may include CP variables supplied as OUT parameters, whose value may be modified as a result of the program's execution.

Each parameterized program must contain a preamble. This preamble processes parameters on behalf of the program's main parameterized subprogram, as follows (see Figure 3-2).

The CP creates a context object for the program and writes the user-specified parameter values into it. The CP then makes the KAPSE call INITIATE_PROGRAM to begin the execution of the specified program using the CP-created context object.

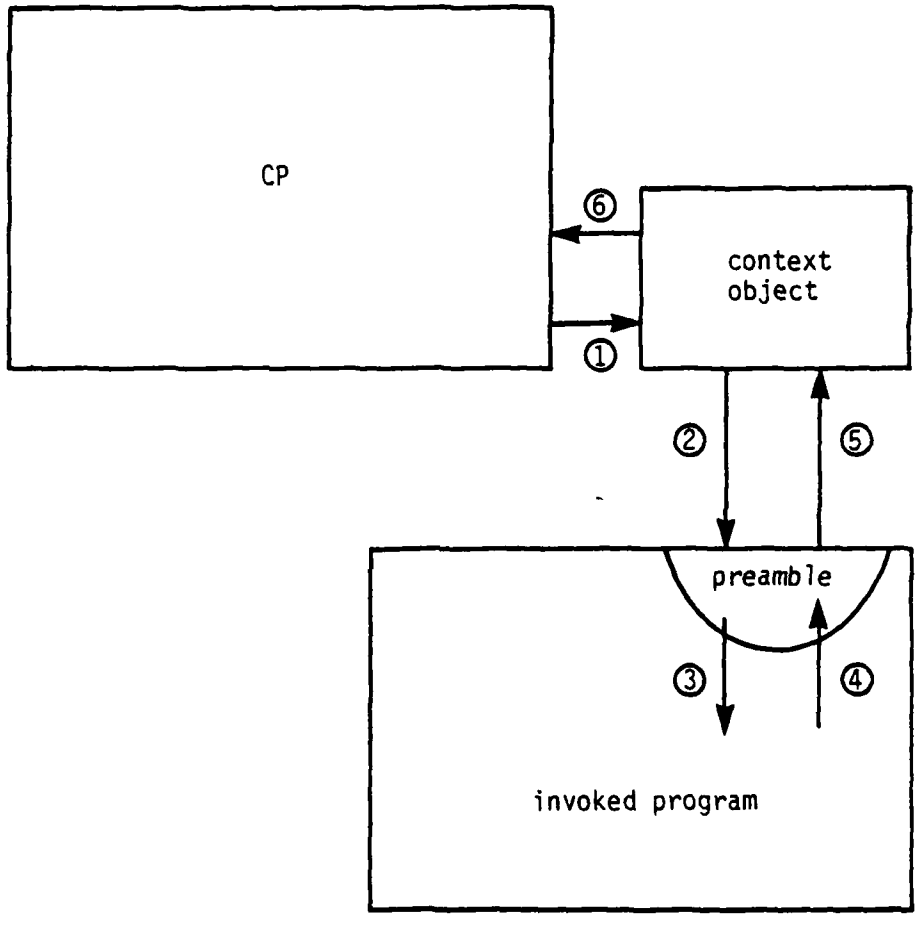
The program's preamble reads the user-specified parameter values from the context object. The preamble then calls the program's main parameterized subprogram, supplying it with the user-specified parameter values. When that subprogram has completed, the preamble writes the updated values of any OUT parameters back into the context object. The CP can then read these updated parameter values from the context object.

A program may also have help information associated with it via "help attributes". These attributes include:

'GENERAL_HELP: The name of a simple object containing general help text for the program. If the program has no general help associated with it, the attribute is undefined.

'PARAMETER_HELP: The name of a composite object containing "parameter help" simple objects. Each parameter help simple object contains help text for a specific program parameter. If no such parameter help text exists, this attribute is undefined.

'OWN_PARAMETER_HELP: If "TRUE", the program interprets actual parameter values of '?' or ':' as requests for parameter help, and supplies this help itself.



21281134-8

FIGURE 3-2: PARAMETER PASSING

3.1.2 Main Constituent Compilation Units

The CP is composed of various Ada subprograms, tasks and packages linked together into an executable program. Referring to Figure 3-3, DRIVER is the main program of the CP. It reads and parses user-typed commands (via PARSE), and passes a parse tree on to the TREE_INTERPRET task to be interpreted.

This interpretation may involve:

1. expression evaluation (EXPRESSION);

2. script parameter manipulation (SUBPROG_SIMULATE);

3. program execution (JOB);

4. activation of other TREE_INTERPRET tasks to process co-routines; or

5. background execution of commands (BACKGROUND).

Any of these actions may manipulate CP variables via the VARIABLE task.

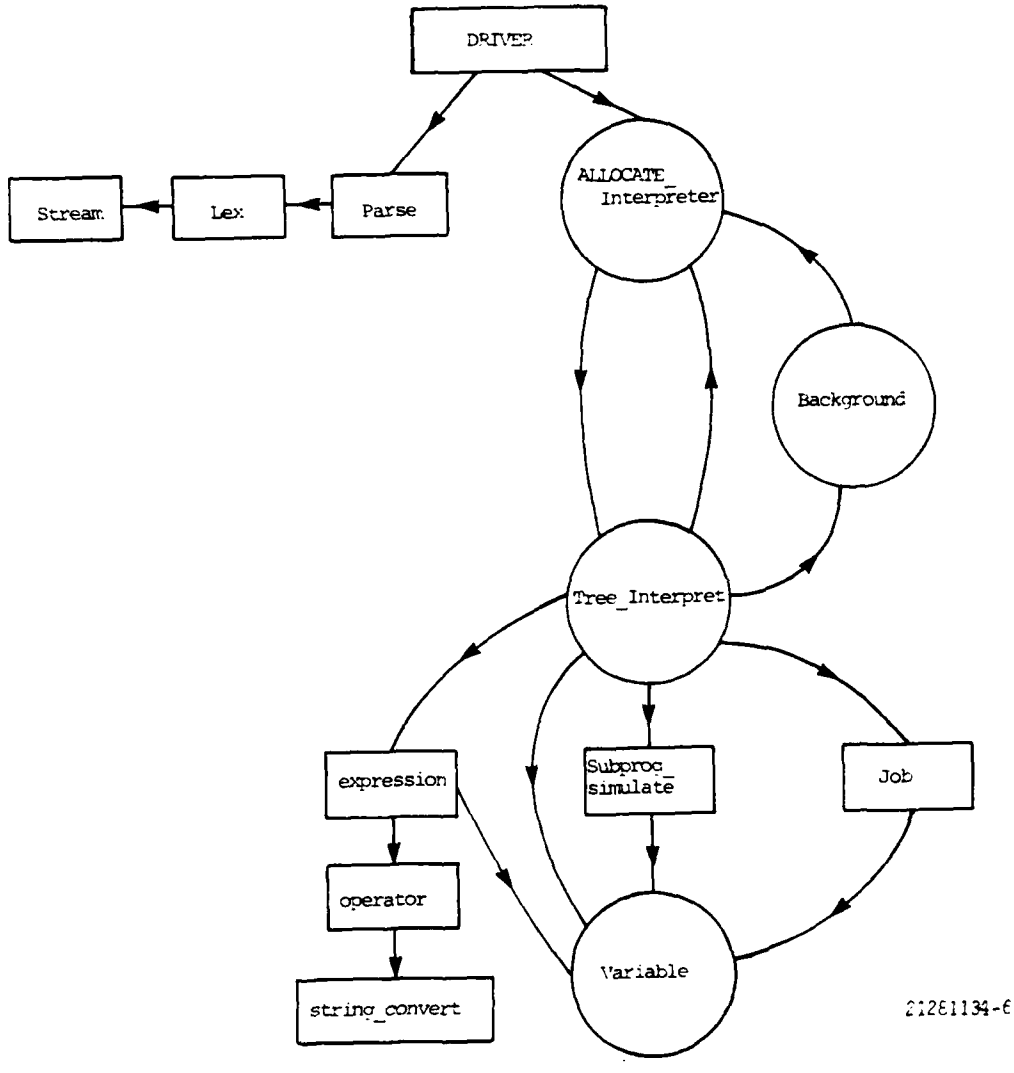


FIGURE 3-3: CP MODULES

Arrows indicate flow of control.
Modules enclosed in rectangles are procedures or packages.
Modules enclosed in circles are tasks.

3.2 Detailed Functional Requirements

3.2.1 The MCL Command Language

The CP interprets commands written in MCL and performs the actions they specify. The MCL language provides commands which facilitate program invocation in an interactive environment. It borrows Ada language features where relevant. For example, the CP expects identifiers supplied to a command to adhere to the Ada syntax for an identifier. Where Ada syntax is inappropriate, new constructs are introduced. Appendix 10.3 summarizes those areas in which MCL syntax differs from that of Ada.

MCL is presented below in two sections. First, the user command repertoire is presented. This is followed by a more formal language description.

The CP command repertoire, summarized in Table 3.1, provides all standard functions required to run MAPSE tools and user programs. Commands can be executed in the foreground or background and can be executed as co-routines. The user can, via the CP, interrupt and restart program execution, manipulate CP variables and database objects, and direct the flow of control of program executions. Commands may be entered interactively or stored, as scripts, for later execution. Via the CP, a user can call any Ada program, including itself. Further, any executing Ada program can call the CP.

In the following discussion, all tokens or constructs that are identical to Ada constructs are so noted with a parenthesized reference to the appropriate Ada LRM section [G-3]. Their formal syntax (expressed via the variant of Backus-Naur Form used in the Ada LRM) appears in the complete syntax in Appendix 10.1. Tokens or constructs that are CP-specific are described in detail and their BNF representation included in the text. Interactions between a user and the CP are included below for illustrative purposes. In such samples, characters printed by the CP are underlined.

3.2.1.1 User Commands

The user requests the services of the CP via commands. A command consists of a sequence of tokens separated by blanks or carriage returns, and delimited by a semicolon (which can be omitted if it immediately precedes a carriage return). A command is optionally preceded by a label, of the form <<identifier>>. Typing the interrupt character (control-C) causes all tokens processed in the current command to be ignored. If a command contains an error, the CP issues an error message.

The CP processes command input line by line; that is, when a carriage return is entered, the line is parsed. Any commands delimited by semicolons are executed. If the tokens immediately preceding the carriage return describe a complete command, the CP assumes that the semicolon was omitted, and executes the command. Otherwise, the CP examines the next line for continuation text.

In interactive mode, a prompt is printed after each line is processed to indicate that the CP is ready to process the next line. This may be a primary prompt (':'), which indicates that the processed line ended with a complete command, or a secondary prompt ("line number/") to remind the user that command input is incomplete.

TABLE 3-1: Command Summary

Command	Purpose	Example
help	provide help for a program or CP script	HELP COMPILE
program call	invoke a program or CP script	COMPILE MYFILE
assignment get put	expression manipulation	%A:= %B+2
create_simple create_composite delete copy rename	manipulate database objects	COPY MYFILE YOURFILE
loop if	flow control	IF %A<2 THEN PUT "OK" END IF
return logout suspend	terminate CP processing	LOGOUT
resume	resume a previously suspended CP session	RESUME MYCP
-|	connect commands as coroutines via pipes	FLIGHT -| LANDER
-> -<	redirect a command's standard input or output	SORT MYFILE -> OUTFILE
SET_INPUT SET_OUTPUT	redirect standard input or output for all subsequent commands	SET_INPUT MYFILE

Table 3-1: Command Summary (Cont'd.)

Command	Purpose	Example
-&	execute a command as a background task	COMPILE MYFILE -&
abort wait	abort a background task or wait for the completion of a background task	ABORT T4
block	group commands for I/O redirection, pipes, or background execution	BEGIN COMPILE A LIST A END -> OUTPUT
stop start cancel status	control the execution of an invoked program or CP script	STATUS .T1.COMPILE
exec	interpret data as a command	EXEC %A & LOW
subprogram simulation	import and export parameters within a script	PROCEDURE COMPLIST (%FILE:STRING) IS BEGIN COMPILE %FILE LIST %FILE END

For example:

```
: COMPILE MYFILE      -- comments as in Ada
: %A :=  2;           -- CP variables begin with a '%'

: COMPILE MYFILE; LIST MYFILE


: <<LISTLOOP>> FOR %I IN 1..5
  1/ LOOP                  -- command not complete yet
  2/   LIST MYFILE END
  3/ LOOP
: -- primary prompt indicates the command is complete
```

Figure 3-4 shows a sample CP session.

```
WELCOME TO THE MAPSE SYSTEM

: HELP COMPILE              -- command help
        .
        .
        .
     help text
        .
        .
        .
: COMPILE MYFILE LIB => MYLIB-- program invocation with
                            -- positional and named parameters

: FOR %I                    -- CP variable names start with '%'

  1/ IN 1..10               -- secondary prompt

  2/ LOOP                   -- flow control.
                            -- LPT is a program which queues text
  3/ LPT MYFILE END LOOP    -- files to be printed on the line printer

: FLIGHTSIM LEVEL=          --another program invocation

: %EXIT_STATUS              -- display exit status of the last invoked
                            -- program
     OK

: LOGOFF
```

FIGURE 3-4: *SAMPLE CP SESSION*

(a) Invoking the CP. When the user logs into the MAPSE, the CP is invoked on his behalf. Associated with this CP is a context object [I-3, 3.2.6.1] that can be used to create temporary database objects, as well as a "current" window (CURRENT_DATA) for permanent database objects. The user may specify a sequence of commands to be performed by the CP as part of its startup, by placing those commands in the database object "CP_STARTUP" in CURRENT_DATA. This startup file enables the user to create a pre-defined environment in which to work. For example, "CP_STARTUP" might contain the text:

```
%LIB:=ADALIB.YOURLIB.MYLIB

CHECK_MAIL                      -- program invocation

"DON'T FORGET TO CALL HOME" -- expression to be printed
```

The CP, as part of its startup, would define the variable %LIB, invoke the CHECK_MAIL program, and print a message. It would then begin taking commands from the user.

The remainder of this section discusses in detail the command repertoire summarized in Table 3-1.

(b) Help Information. Information about invokable programs may be obtained via the HELP command.

```
help_command ::= HELP [database_literal]
```

HELP with no parameter requests general information about all CP commands and invokable programs. More information about a specific program can be requested by providing the program name as an argument to HELP. If the program specified does not exist or help is not available for it, the user is informed. Notice that a program name is specified as an unquoted string. Within commands, the CP interprets program names in terms of the PROGRAM_SEARCH_LIST attribute associated with the CP's context object. The program must lie within one of the composite objects named in PROGRAM_SEARCH_LIST.

(c) Program Invocation. A user can invoke programs and functions with or without parameters.

PROGRAM CALLS: The program call command invokes an arbitrary linked executable object.

```
program_call_statement ::= objname[actual_parameter_part]
actual_parameter_part ::=
  [(]parameter_association {separator parameter_association}[)]
separator ::= blank | comma
parameter_association ::=
  [formal_parameter=>] actual_parameter
formal_parameter ::= identifier
actual_parameter ::= expression | help_mark
help_mark ::= ? | :
```

The syntax shown above for a program call is similar to Ada syntax for a procedure call, except that: (1) the procedure name is replaced by the name of the program to be invoked; (2) the parentheses surrounding the actual parameter part may be omitted; and (3) the comma between parameter associations may optionally be replaced by a blank. However, if the user wishes to continue the parameter list on the next line, a comma is required after the last parameter association on the current line.

PARAMETER PASSING: The actual parameter part of a program call consists of one or more parameter associations, each of which specifies an actual parameter to be passed to the invoked program, either positionally or by name.

For positional parameters, the actual parameter corresponds to the formal parameter with the same position in the invoked program's formal parameter list. For named parameters, the corresponding formal parameter is explicitly given in the call [G-3, 6.4]. Unlike in Ada, positional and named parameters may be freely mixed. The CP parses the parameter association by grouping together all positional parameters, followed by all named parameters. For named parameters, the CP convention is that only the last occurrence of a parameter association is used, permitting a user to change a mistyped named parameter without retyping others. For example, the line:

```
: FLIGHT  3 5 LEV=>2 6 7 LEV=>3
```

associates the value 3 with LEV.

An invoked program may associate a mode of IN, OUT, or IN OUT with a formal parameter. An actual parameter of mode IN OUT or OUT must be represented by a variable name. If it is not, a warning is printed, and the parameter is ignored. The specified variable's value is updated when the program completes its execution.

As in Ada, if a program's declaration specifies a default value for an IN parameter, then the corresponding parameter may be omitted from a call [G-3, 6.4.2].

MCL also implements default OUT parameters if the user fails to specify an actual parameter of mode OUT, or if the specified parameter was not a variable name. In these cases, the CP generates an implicit variable declaration. The generated variable's name is the catenation of '%' and the formal parameter name. For example, if a program COMPILE's main subprogram had the specification:

```
procedure compile (file: string; lib: string;
                   max_error_severity:  out string);
```

the CP user could type

```
: COMPILE MYFILE MYLIB
```

At the conclusion of COMPILE's execution, a CP variable named %MAX_ERROR_SEVERITY would be generated and assigned the OUT parameter value. The CP user is informed of any variables generated via default OUT parameters if the pre-defined CP variable %INFORM_DEFAULT_OUT is TRUE (Section 3.2.1.2).

A default OUT parameter may generate a variable name which conflicts with a currently existing CP variable. For example,

```
: %MAX_ERROR_SEVERITY:= OK
: COMPILE MYFILE MYLIB
: --default OUT parameter variable
: --%MAX_ERROR_SEVERITY is already defined
```

If the pre-defined CP variable %AUTO_REDEFINE is TRUE, the OUT parameter value replaces the current value of the variable. If %AUTO_REDEFINE is FALSE, the user is queried as to whether the already defined variable should take on its new default OUT parameter value.

PARAMETER HELP: The user may request information about a positional or named parameter by supplying a question mark in place of the actual parameter value, followed by a new line. Since the '?' is not part of the basic graphic character set, a colon may be used instead. The action taken depends on the manner in which the invoked program chooses to deal with requests for parameter help. If the program can provide no parameter help, the user is informed, and may continue specifying parameter associations to the program. For example:

: MYPROG 2 ?

No help available for MYPROG. Please continue supplying parameters:

If the called program has associated with it help text for the relevant parameter, then that text is printed, and the user may continue specifying parameters. If the called program wishes to interpret any help requests itself, the actual parameter part is considered complete, and the program is invoked. At the conclusion of the program's invocation, the user is informed that no further parameter associations will be accepted but, rather, the program call command must be resubmitted. For example:

```
: FLIGHTSIM LEVEL --> ?
    .
    .
    .
dialogue with the invoked program FLIGHTSIM
    .
    .
    .
Please resubmit the program call
:
```

FUNCTION INVOCATION: An MCL function is a program that returns a value. The syntax for a function invocation is similar to that for program calls:

function_call ::= objname([actual_parameter_part])

A function call's actual parameter part is similar to that of a program call, the differences being that it must be surrounded by parentheses and that it can only contain IN parameters. The user may also request parameter help for a function.

Examples of function calls:

```
LEXICAL_LENGTH(%V)

SINE (2.0)
```

(c) Expression Manipulation. MCL commands are provided to:

1. store an expression's value in a variable or database attribute (assignment);
2. read and store a user-typed literal (GET), and;
3. display the value of an expression (PUT).

ASSIGNMENT: The assignment command replaces the current value of a variable or database object attribute with the value of a string.

```
assignment_statement ::=

        attribute := expression |

        variable  := expression
```

The right-hand side of an assignment must be a a string. If it is not, an implicit conversion is performed.

Since all CP options are controlled via pre-defined variables (Section 3.2.1.2), the assignment statement can be used to modify these options.

Examples:

```
: %V1:="HELLO"

: %V1:=HELLO                        -- equivalent to above,
                                    -- since identifiers don't
:                                   -- have to be surrounded
                                    -- by quotes.

: .MYLIB.MYFILE'CONFIG := 4         -- an attribute (named
                                    -- CONFIG) of the
                                    -- database object
                                    -- MYLIB.MYFILE
: %PROMPT := "%"                    -- modify CP prompt

%
```

GET: The GET command causes the CP to read an arbitrary sequence of literals from standard input and store them (as strings) in CP variables.

```
get_command ::=  GET variable {separator variable}
```

The literals must describe values of valid MCL types, and must be separated by blanks, commas, or newlines. Examples:

```
: GET %V1 %V2       -- read values from standard input

     "V1's VALUE"

     2.0

: GET %V1 %V2 -< INFILE

:                   --redirect GET's input. INFILE must contain
                    --two literals
```

PUT: The PUT command causes the CP to print the values of an

arbitrary sequence of expressions to standard output.

```
put_command ::=  PUT expression {separator expression}
```

For example:

```
: PUT %A %B

: PUT "THIS STRING IS PRINTED" & " TO STANDARD OUTPUT"

: PUT HELLO      -- unquoted string
```

The keyword "PUT" may be omitted when a single expression is to be displayed except in the case of an unquoted string, which is assumed to be a program call. For example:

```
:.T1.COMPILE'EXECUTION_TIME --attribute value which gives
                            -- execution
  5.2                       --time of an invoked program

: %X = %Y

  FALSE
```

(e) Database Manipulation. A variety of KAPSE primitives are available as MCL commands. The list below is representative of commands for database manipulation. See [I-3, 3.2.1] for a complete list of database manipulation facilities.

```
contents(name: in string) -- returns the entire contents of a text
                          -- simple object.

create_simple (name: in string); -- creates a simple object.

create_composite (name: in string; component_da: in string);
                                   -- creates a composite object.

create_window (name: in string;
               target: in string;
               common_ancestor: in string :="";
               partition: in string :="";
               capacity: in string :="");
    -- creates a window.

delete (name: in string);
    -- Deletes a simple object, composite object or window.


copy (oldname: in string; newname: in string);
     -- Copies a simple object, composite object or window.
```

```
revoke (super_window: in string;
        sub_window: in string);
     -- Incapacitates he specified sub_window if it was derived from
     -- the specified super_window.

get_all_attributes (name: in string);
     -- Returns the value of all  non-null non-distinquishing
     -- user-defined attributes of a database object.
```

Examples:

```
: CREATE_WINDOW  MYWINDOW YOURWINDOW
: CREATE_COMPOSITE MYWINDOW.YOURWINDOW.LOG "MODULE RELEASE_NUM"
: COMPILE MYFILE -> MYWINDOW.YOURWINDOW.LOG.COMPLOG
: DELETE MYWINDOW.YOURWINDOW
: GET_ALL_ATTRIBUTES(MYWINDOW)

      (PURPOSE=>TESTING, CHECKLEVEL=>2)
```

(f) Flow Control. MCL commands can be used to affect the flow of control of program invocations.

IF: The IF command selects for execution one or none of a sequence of MCL commands, depending on the value of one or more corresponding conditions.

The syntax for the IF command is identical to Ada's [G-3, 5.3]:

```
if_statement ::=

        IF condition THEN command_input

           {ELSIF condition THEN command_input}

           [ ELSE command_input] END IF

condition ::= expression
```

The expressions specifying conditions must be of the pre-defined type BOOLEAN.

LOOP: A loop command specifies that a sequence of statements in a basic loop is to be executed zero or more times. Its syntax is identical to Ada's [G-3, 5.5].

```
loop_statement ::=
        [iteration_clause] basic_loop
iteration_clause ::=
        FOR loop_parameter IN discrete_range |
        WHILE condition
loop_parameter ::= variable
discrete_range ::= expression..expression
basic_loop ::= LOOP command input END LOOP
```

The loop command, like all CP commands, may be executed in interactive as well as batch mode.

An exit command causes the termination of an enclosing loop.

```
exit_statement ::= EXIT
```

An exit command may only appear within a loop [G-3, 5.7]

A loop command without an iteration clause specifies repeated execution of the basic loop. The basic loop may be left via an interrupt, via an exit command, or if a command within it terminates the CP session.

If a loop command contains a WHILE clause, the condition is evaluated and tested before each execution of the basic loop. If it evaluates to FALSE, the loop statement terminates; otherwise, the basic loop is executed.

If a loop command contains a FOR clause, the loop parameter and discrete range are re-evaluated and tested before each execution of the basic loop. Examples:

```
: FOR % I IN 1...4  LOOP
  1/ GET % A
  2/ IF % A /= ""
  3/ THEN FLIGHTSIM %A
  4/ END LOOP
```

(g) Terminating CP Processing. There are several ways to terminate CP processing. In all cases, the CP must be quiescent (i.e., no background tasks can be active); otherwise, the user is warned and the terminating command is ignored.

RETURN: The RETURN command terminates the execution of the CP, and returns control to the CP's invoker.

return_statement ::=

RETURN [expression]

If the CP is processing a script that describes a function (Section m, below), the RETURN command must include an expression whose value is the result returned by the function.

LOGOUT: The logout command also terminates the execution of the CP.

logout_statement ::= LOGOUT

Unlike RETURN, however, the LOGOUT command causes the termination of the entire MAPSE session. Control is returned to the system initiator.

SUSPEND: The SUSPEND command terminates CP processing, maintaining its current context, and returns control to the invoker.

suspend_statement ::= SUSPEND objname

The argument, objname, specifies a database object in which the CP's context is to be saved for reference by a subsequent RESUME command.

: *SUSPEND MYCP*

CP SESSION SUSPENDED

MAPSE SESSION TERMINATED -- typed by the CP's invoker which

-- was the System Initiator

SHUTDOWN COMMANDS: The user may specify a sequence of MCL commands to be performed automatically prior to CP termination via LOGOUT or RETURN. These commands must be contained in the database object "CP_SHUTDOWN" in the CP's CURRENT_DATA window.

For example, "CP_SHUTDOWN" might contain the text

```
STATUS %TERMINATED_JOBS
LIST MYFILE
```

RESUME: The RESUME command resumes a previously SUSPENDed CP session.

resume_command::=RESUME objname

The jobname argument to RESUME is the name under which the context of a previously SUSPENDed CP was stored.
Example:

```
: RESUME MYCP

: --this prompt is printed by the resumed CP session.
```

(h) I/O Redirection. Many CP commands (for example, PUT) read from standard input or write to standard output. If unmodified, a command's standard I/O defaults to the CP's own standard input and output. The user may redirect the standard input or output of any command via several convenient notations.

PIPES: The notation -| between two commands indicates that the commands are to be connected via a pipe [I-3,3.2.3.1]. Standard output of the command to the left of the -| becomes the standard input of the command to the right of the -| and the commands execute as co-routines. A sequence of commands connected via pipes is referred to as a pipeline command.
For example:

```
: "STRING TO SERVE AS STANDARD INPUT" -| LPT
                --string is queued to
                --be printed by line printer

: DATE -| FLIGHT_LOG OPTION => HEADER
```

STANDARD I/O: The notation -< is used to redirect a command's standard input. The notation -> is used to redirect a command's standard output. The specified database object is opened (for standard input) or created (for standard output). The command is interpreted, and the database object is closed. For example:

```
: FLIGHTSIM -< CONTROL_FILE ->RESULTS
                --FLIGHTSIM reads its standard input from
                --CONTROL_FILE and writes its standard output
                --to RESULTS

: COMPILE MYFILE LIB=>MYLIB->RESULTS
```

Notice that the second output redirection to RESULTS caused the first command's output to be overwritten.

SET_INPUT and SET_OUTPUT: The notations -< and -> modify the standard input and output of a single command. The commands SET_INPUT and SET_OUTPUT modify the default standard input and output for all commands executed by the CP.

```
set_input_statement ::=  SET_INPUT [objname]

set_output_statement ::=  SET_OUTPUT [objname]
```

Each takes as an argument the name of the database object which is to serve as standard input (output) for subsequent commands . The database object is opened (for standard input), or created (for standard output), and remains open until a subsequent SET_INPUT (SET_OUTPUT) call, or until the CP session terminates. If the database object name is omitted, standard I/O for commands reverts to

the CP's standard input or output.
For example:

```
: SET_OUTPUT  .TMP

: LIST MYFILE

: LIST YOURFILE
```

would cause .TMP to contain a concatenated listing of MYFILE and YOURFILE. Standard output of any other command would also be appended to .TMP until the session terminated, or until the user reset the default command output. For example:

```
: SET_OUTPUT -- reset to CP's standard output
```

SET_INPUT causes all subsequent commands to take their input from the specified database object. If the end of the input file is reached, the user is informed, and default command input is automatically reset to standard input. For example, if .TMP contained the string

```
"10 TRUE HELLO 1.0"
```

the user might type

```
: SET_INPUT .TMP

: GET %A            -- 10

: GET %B %C         -- TRUE and HELLO

: GET %D            -- 1.0

: SET_INPUT         -- reset to CP's standard input
```

(i) Background Execution. An MCL command is normally executed in the foreground; i.e., the CP waits for the command to complete its execution before accepting further command input. The user may specify that a command is to execute in the background via the -& notation. In this mode, the CP begins the execution of the command, but does not wait for it to terminate before accepting subsequent user commands. For example:

```
: COMPILE MYFILE LIB => MYLIB -&
```

A background command can be viewed as a task object that executes asynchronously. As in Ada, this object has a name, of the form "TX", where X is an integer incremented for each task generated. Alternatively, if the background command is preceded by a label, the task name is equivalent to the label name. In either case, the name

is displayed to the user when the task begins its execution, and is also stored in the pre-defined CP variable %LAST_TASK.
For example:

```
: COMPILE MYFILE LIB =>MYLIB -&

  T4 executing

: -- ready to accept next command

: <<MYTASK>> FOR %I in 1..4 LOOP

  1/ FLIGHTSIM %I END LOOP -&

  MYTASK executing
:
```

The user is informed when the task completes its execution.
For example:

```
MYTASK completed
:
```

The task name can be used to refer to the task in order to abort it, via the ABORT command.

abort_statement ::= ABORT task_name {separator task_name}

task_name ::= identifier

The specified background tasks are aborted.

The user may begin the execution of a command in the background, then decide to wait for it to conclude its exectuion. The WAIT command causes the CP to wait for the conclusion of the specified task(s) before accepting further command input from the user.

wait_statement ::= WAIT task_name {separator task_name}

For example:

```
: COMPILE MYFILE LIB =>MYLIB -&

  T4 executing

: ABORT %LAST_TASK

  T4 aborted


: <<COMP>> COMPILE YOURFILE -&

  COMP executing

: WAIT COMP
```

COMP completed

:

A pipeline command may be invoked in the background. For example:

: FLIGHTSIM -| SORT -&

T6 executing

:

Any OUT parameter variables in a background command are updated as the command executes. The user should avoid referencing these variables before the command terminates, since their value is indeterminate.

The commands SET_INPUT and SET_OUTPUT within a background command reset default input and output only for the background command. Foreground defaults are unaffected.

(j) Blocks. Multiple commands may be grouped together, in order to apply a notation to all of them, by means of the block command.

block_statement ::=

BEGIN command_input END

For example, the block

: BEGIN LIST MYFILE

1/ COMPILE MYFILE LIB =>MYLIB

2/ END -| LPT

is equivalent to

: LIST MYFILE -| LPT

: COMPILE MYFILE LIB =>MYLIB -| LPT

The block command may be applied to -< and ->. The specified database object is opened (for standard input) or created (for standard output) and remains open for the duration of the block. Thus,

: SET_OUTPUT .TMP

: LIST MYFILE

: LIST YOURFILE

: SET_OUTPUT

is equivalent to

```
: BEGIN

   1/ LIST MYFILE

   2/ LIST YOURFILE

   3/ END - > .TMP
```

except that each command in the first example is executed as it is typed, while commands in the block are not executed until the entire block has been entered.

The block command also allows the user to group together commands for sequential background execution. For example:

```
: BEGIN

   1/ COMPILE MYFILE LIB =>MYLIB

   2/ FLIGHTSIM

   3/ END -&

   T12 executing

:
```

causes *COMPILE to* be executed in the background, followed by FLIGHTSIM. Notice that this is different from:

```
: COMPILE MYFILE LIB =>MYLIB -&

   T12 executing

: FLIGHTSIM -&

   T13 executing
```

in which COMPILE and FLIGHTSIM execute simultaneously.

(k) Jobs. Some MCL commands are executed via internal CP actions (e.g., assignment) while others cause the CP to invoke external programs. This latter category includes a program or function call and a script invocation. A command in this category is referred to as a job. Each invoked program has associated with it a context object, which can be referenced to control the program's execution and determine its status.

CONTEXT OBJECT NAMES: Each background task creates, within the CP's context object, a composite object whose name corresponds to the task name. Within this composite object, the task creates a context object for each program it invokes. The name of each context object

is equivalent to the name of the invoked program. If the same program is subsequently invoked within the same task, its context object name is of the form "program_nameX", where X is an integer starting from 2 and incremented for each repeated invocation of the program within the task. For example, the commands:

```
: COMPILE MYFILE LIB=>MYLIB -&

  T4 executing

: <<COMP>> BEGIN COMPILE MYFILE

  1/ COMPILE YOURFILE END -&

  COMP executing

:
```

would cause the creation of composite objects named ".T4" and ".COMP", and context objects named ".T4.COMPILE", ".COMP.COMPILE" and ".COMP.COMPILE2".

Foreground CP execution can be thought of as a task object named "T1" which interacts with the user and executes commands synchronously. Context objects for any foreground programs are thus created in the composite object ".T1". The pre-defined CP variable %FCONTEXTS contains a concatenated list of context object names created in ".T1" for the last foreground job. For example:

```
: COMPILE MYFILE LIB=> MYLIB

: %FCONTEXTS

    .T1.COMPILE
```

The pre-defined CP variable %ACTIVE_CONTEXTS contains a concatenated list of context object names for all active programs.

JOB CONTEXT OBJECT ATTRIBUTES: A program's context object has various attributes that give information about the program's execution, as shown in Table 3.2. The context object may be treated as a database object for the purpose of referencing these attributes.

TABLE 3.2 JOB ATTRIBUTES

Attribute Name	Possible Values	Meaning
TERMINATED	"TRUE" "FALSE"	Indicates whether the program's execution has completed.
EXECUTION_TIME	a string describing a REAL	Total execution time for the program (defined only if TERMINATED = TRUE)
EXIT_STATUS	"OK", "CANCELLED" "INTERRUPTED", or the name of an unhandled exception	The program's exit status (defined only if TERMINATED = "TRUE")

The pre-defined variables %EXECUTION_TIME and %EXIT_STATUS contain the execution time and exit status for the last foreground job. If a program completes its execution with an exit status of "OK", all database objects within its context object are deleted.

Examples:

```
: COMPILE MYFILE LIB =>MYLIB

: %EXIT_STATUS

    OK

: COMPILE YOURFILE LIB =>MYLIB -&

  T7 executing

: .T7.COMPILE'TERMINATED

  FALSE
```

CONTROL COMMANDS: The MCL context object commands, describe below, take as arguments a sequence of context object names or the names of composite objects containing context objects, and perform actions on the programs they represent.

STOP: The STOP command is invoked to stop the execution of an arbitrary executing background job. STOP takes as its argument one or more context object names.

```
stop_command ::= STOP job_list

job_list ::= job_name {separator job_name}

job_name ::= expression
```

START: The START command allows the user to restart the execution of a job suspended by a prior STOP command.

```
start_command ::= START job_list
```

The specified program(s) resume their execution in the background.

CANCEL: The CANCEL command is invoked to terminate the execution of one or more background jobs.

```
cancel_command ::= CANCEL job_list
```

STATUS: The STATUS command enables the user to query the status of programs.

```
status_command ::= STATUS job_list
```

If a program within the job list is currently active, the user is given the following information about its execution: execution time, memory used, I/O count. If the program is terminated, its total execution time and exit status are displayed.

Examples:

```
: <<F>>FLIGHTSIM -| LANDER -&
  F executing
: STOP .F.FLIGHTSIM
: DEBUG  .F.FLIGHTSIM
        .
   debugging
        .

: START .F.FLIGHTSIM
: STATUS .F.FLIGHTSIM
        .
        .
        .
   status information
        .
        .
        .
: CANCEL .F --all context objects
            --within .F, i.e,
            --.F.FLIGHTSIM and
            --.F.LANDER
```

EXEC: The CP normally reads user-typed strings and interprets them as commands. The CP is also capable of interpreting data as a command, via EXEC.

```
exec_statement ::= EXEC expression
```

EXEC takes as its argument an expression that must yield a string value that is recognizable as one or more MCL commands. For example:

```
: %A := "COMPILE MYFILE LIB=>"

: EXEC %A & MYLIB -- the command COMPILE MYFILE LIB=>MYLIB is

:                 -- executed

:  EXEC %A & YOURLIB -- COMPILE MYFILE LIB=> YOURLIB
```

EXEC's can be nested. For example

```
: %A:= "EXEC ""BEGIN COMPILE MYFILE;"" & %B &""MYFILE END->.TMP"""

: %B:= LIST

: EXEC %A -- BEGIN COMPILE MYFILE; LIST MYFILE END ->.TMP

: %B:= DELETE

: EXEC %A -- BEGIN COMPILE MYFILE; DELETE MYFILE END ->.TMP
```

(1) Nested CP's. When the user logs in to the MAPSE System, the CP may be invoked on his behalf. This CP is capable of invoking an arbitrary program, including the CP itself. For example:

```
: CP

: -- This prompt is typed by the nested CP.
```

The user may then issue commands to the invoked CP, which is referred to as a nested CP. The nested CP has its own context object,and any changes made to variables in the nested CP are not reflected in the invoking CP.

If a nested CP session is terminated by the CP commands RETURN or SUSPEND, control is returned to the invoking CP. If the LOGOFF command is issued, the MAPSE session is terminated (including the invoking CP).

A nested CP inherits the standard input and output of the invoking CP. It reads commands from its standard input, and commands may in turn read from standard input or write to standard output as part of their execution. As with any invoked program, a nested CP's I/O may be redirected via -> or -<. If the CP's standard input is

not the teletype an end-of-file is equivalent to RETURN. For example, if a database object CP_INFILE contained the text.

```
FOR %I IN 1..5 LOOP GET%A; COMPILE%A; LIST%A END LOOP;

MYFILE YOURFILE F1 F2 TESTFILE -- This data in
                               -- standard input will be read
                               -- by GET
```

the user might type

```
: CP -< CP_INFILE
```

The CP can optionally be supplied with a string which is to serve as its standard input, via the parameter INPUT_STRING. For example:

```
:  CP INPUT_STRING => "COMPILE MYFILE; %EXIT_STATUS"
```

Since the CP is an ordinary tool, it can be invoked from any other program. The INPUT_STRING parameter provides a convenient means of invoking the CP within a program, specifying to it the command(s) it is to execute and data for those commands.

(m) CP Scripts. A script is a sequence of CP commands stored in a database object. It is functionally equivalent to a linked executable program in terms of invocation syntax, help information and parameter passing. Any command within the script may read from standard input or write to standard output. For example, if a script named "SIM" contained the text.

```
"HOW MANY TIMES SHOULD THE SIMULATION BE RUN?"

GET %TIMES

For %I IN 1..%TIMES LOOP

     FLIGHTSIM OPTION=>FAST LEVEL=>3

END LOOP
```

the user might type

```
: HELP SIM
      .
      .
      .
   help text
      .
      .
      .

: SIM
```

```
HOW MANY TIMES SHOULD THE SIMULATION BE RUN?

10
: -- The script has completed its execution.
```

A script differs from an input file for a nested CP in that it contains only commands. The data for these commands comes from standard input, not the script. This is in keeping with the model of scripts as equivalent to linked executable programs.

A script can have its I/O redirected or be executed in the background. For example, if a database object "SIMIN" contained the text "10", the user might type:

```
: SIM -< SIMIN -&
```

A script's exit status and execution time are available to the user in a manner identical to that for programs. For example:

```
: SIM -< SIMIN

: %EXIT_STATUS

  OK
```

A script is terminated when end-of-file is reached, or when a RETURN, LOGOUT or SUSPEND command is encountered.

SUBPROGRAM SIMULATION: A script may receive IN parameter values and return updated OUT parameter values if it contains subprogram simulation command.

```
subprogram_simulation_statement ::=

    subprogram_specification IS subprogram_simulation_body

subprogram_simulation_body ::= BEGIN command_input END [identifier]

subprogram_specification::=

    PROCEDURE identifier [formal_part] |

    FUNCTION  identifier [formal_part]   RETURN subtype_indication

formal_part::=(parameter_declaration{semi_colon parameter_declaration})

parameter_declaration::=

   CP_identifier_list :mode subtype_indication  [:=expression ]

CP_identifier_list::=  variable {, variable}

mode::=  [IN] | OUT | IN OUT
subtype_indication::=  INTEGER | REAL | BOOLEAN | STRING
```

The subprogram simulation command simulates the execution of a parameterized Ada subprogram. It consists of a subprogram specification that gives parameter information, and a subprogram simulation body, which contains CP commands to be interpreted in order to simulate the subprogram's execution.

The subprogram specification is similar to an Ada subprogram specification [G-3, 6.1]. It may specify either a procedure or a function. Parameter declarations are separated by semicolons. Parameters and function return values must be of one of MCL's pre-defined types (Sec. 3.2.1.2). Each formal parameter name must follow the format of a CP variable. An example of a subprogram simulation statement is:

```
Procedure COMPILE_AND_LIST (%FILE:STRING;
                            %LIB: STRING;
                            %COMPILE_STATUS: out STRING)

IS
  BEGIN
   %DEBUG:= "LOOP GET %C; EXEC %C;END LOOP"
   COMPILE %FILE LIB=>%LIB
   %COMPILE_STATUS := %EXIT_STATUS
   EXEC %DEBUG --reads and executes user
               --commands until an EXIT command.
               --allows the interactive user to
               --examine script variables, etc.
   IF %COMPILE_STATUS = OK THEN LPT %FILE
  END
```

If this text were placed in a database object named COMPLIST, the CP user could invoke it by typing

```
: COMPLIST MYFILE MYLIB
```

Treatment of parameters is identical to that for program invocation. The script receives parameters and performs type checking based on the parameter specification. If an actual parameter is not of the proper type, and cannot be implicitely converted to the proper type, the script's execution is terminated. Otherwise, the script performs the commands in its subprogram simulation body, and returns updated values of OUT parameters. If a script describes a function, the function return value must be specifed a RETURN command. As in program invocation, default OUT parameters are generated on behalf of the user. In the example above, a variable named %COMPILE_ERRORS would be generated.

AFFECTING THE CP'S ENVIRONMENT: A script is functionally equivalent to a program, and cannot directly modify the invoking CP's environment. For example, any variables declared in a script cannot be subsequently referenced in the invoking CP. The user may specify that the current CP should execute a sequence of MCL commands contained in a database object via the EXEC command. For example, if a database object named "VARIABLES" contained the text:

```
%MYLIB := ADALIB.YOURLIB.MYLIB

%MYFILE := ADALIB.SYSLIB.TESTFILE
```

the user might type

```
: EXEC CONTENTS(VARIABLES)

: --variables defined in VARIABLES are now visible

: --in the current environment

%MYLIB

  ADALIB.YOURLIB.MYLIB
```

3.2.1.2. Language Elements

(a) Lexical Elements. CHARACTER SET : Identical to Ada [G-3, 2.1]. This includes a basic graphic character set, as well as other characters from the ASCII graphics set. Any command can be expressed using only the basic character set. Any lower-case letter is equivalent to the corresponding upper-case letter except within character strings and unquoted strings.

LEXICAL UNITS: An MCL input stream is a sequence of lexical units. The lexical units are identifiers (including reserved words), numeric literals, character literals, strings, delimiters, and comments.

A delimiter is either one of the following special characters in the basic character set:

```
&'()*+,-./:;<=>
```

one of the following compound symbols:

```
=> ..  :=  /= >= <= -> -< -| -&
```

or the character:

```
?
```

from the ASCII graphics set. Adjacent lexical units may be separated by spaces or by passage to a new line. An identifier or numeric literal must be separated in this way from an adjacent identifier or numeric literal. Spaces must not occur within lexical units, except within strings and comments. Each lexical unit must fit on one line. (G-3 Sec. 2.2].

IDENTIFIERS: MCL identifiers are identical to Ada identifiers [G-3, 2.3]. As in Ada, identifiers differing only in the use of corresponding upper and lower case letters are considered to be the same. Examples:

```
INT1    Ada_LRM
```

NUMERIC LITERALS: There are two classes of numeric literals: integer literals and real literals. Integer literals are the literals of the MCL type INTEGER. Real literals are the literals of the MCL type REAL. As in Ada, isolated underscore characters may be inserted between adjacent digits of a decimal number, but are not significant. [G-3,2.4]. The conventional decimal notation is used. Real literals are distinguished by the presence of a decimal point.

An An exponent indicates the power of 10 by which the preceding *number* is to be multiplied to obtain the value represented. An integer literal can have an exponent; the exponent must be positive or zero.

Examples:

```
12 0 123_4561E6 --integer literals

12.0  0.0  0.456  3.14159_26  --real literals

12.34E-4   --real literal with exponent
```

CHARACTER STRINGS: Identical to Ada [G-3, 2.6].

Examples:

```
""      -- empty string

"ABC"

""""    --a single included string bracket character
```

DATABASE LITERALS: Database literals name database objects. There are two classes of database literals: partition specifiers and object names [I-3, 3.2.1]. The user may specify database literals using full attribute value notation or shorthand positional notation The BNF syntax for a database literal is given below:

```
database_literal ::= partition | objname

partition ::= [.]id_or_av_list_or_star{.id_or_av_list_or_star}

objname ::= [.]id_or_av_list {.id_or_av_list}

id_or_av_list_or_star::= identifier | attribute_value_list

id_or_av_list ::= identifier | attribute_value_list

attribute_value_list ::= (attribute_association
                                    {comma attribute_association}

attribute_association ::= [formal_attribute_name=>]expression

formal_attribute_name ::= identifier
```

Notice that an identifier is a database literal.

The first character of a database literal indicates the desired window. If the first character is a ".", the database literal is rooted in the CP's context object. Otherwise, the database literal is rooted in the CURRENT_DATA window associated with each user via the KAPSE.

A database literal beginning with a "." has different meanings to different program invocations, since each invocation has its own context object. However, each invoked program does automatically receive a window on the context object of its invoker, its invoker's invoker, and so on, back to the first program invoked by the system initiator. Therefore, a user-typed database literal beginning with a "." automatically has the name of the window on the CP's context object appended to it by the CP. This process, referred to as normalizing the database literal, allows the literal to be passed to an arbitrary invoked program without changing its meaning.

The CP's context object is automatically deleted at the conclusion of the CP session. Any database object whose name begins with a "." is thus a temporary object that will be deleted along with the context object. For example:

```
TEST            -- a database object named "TEST"
                -- located in the CURRENT_DATA window

.TEST           -- a temporary database object named "TEST"
                -- If the window on the CP's context
                -- object were named
                -- "F4", this database literal would be
                -- normalized into  ".F4.TEST"
FLIGHT.SIM.*    -- a partition

FLIGHT.(RELEASE=>3,MODULE=>STICK)  --attribute value list
                                   --notation
```

COMMENTS: identical to Ada.

Examples:

```
-- comments have no effect on the meaning of commands;
-- their sole purpose is the enlightenment of the human
-- reader
-- [G-3, 2.7)
```

RESERVED WORDS

The following keywords are reserved in MCL, and may not appear as the name of a program or script to be invoked. However, they may appear as identifiers anywhere else in the command line:

TRUE, FALSE,GET,PUT,IF,THEN,ELSIF,ELSE,BEGIN,END,FOR,WHILE,LOOP, EXIT,RETURN,LOGOUT,SUSPEND,RESUME,ABORT,WAIT,HELP,SET_INPUT, SET_OUTPUT,PROCEDURE,FUNCTION,EXEC,STOP,START,CANCEL,STATUS, LIST_VARS.

(b) Types. MCL contains no facility for declaring types. A CP literal must be one of the following pre-defined types:

1. INTEGER - This pre-defined type is implemented as in Ada [G-3, 3.5.4]. INTEGER values are expressed as integer literals.

2. REAL - This pre-defined type describes floating point real numbers. Its Ada declaration is

   ```
   TYPE REAL is digits NUM_DIGITS
   ```

   where NUM_DIGITS is within the range of the most accurate numeric type supported by the implementation. REAL values are expressed as real literals.

3. STRING - The pre-defined type STRING denotes an unconstrained, one-dimensional array of characters. Its Ada declaration is

   ```
   TYPE STRING is array (NATURAL range<>) of CHARACTER;
   ```

   STRING values are expressed as character strings. Catenation is a pre-defined operator for strings; it is represented as &. The relational operators <, <=, >, and >= are defined for strings, and correspond to lexicographical order.

4. BOOLEAN - As in Ada, there is a pre-defined enumeration type named BOOLEAN [G-3, 3.5.3]. It contains the two literals FALSE and TRUE ordered with the relation FALSE<TRUE.

5. UNQUOTED_STRING - The pre-defined type UNQUOTED_STRING denotes a string that has the lexical form of a database literal (with the exception of TRUE and FALSE, which are BOOLEAN enumeration literals).

   Unlike an MCL string, an unquoted string does not have to be surrounded by string bracket characters. This provides the user with a convenient means of specifying identifiers or database object names without having to quote them, as in

   ```
   HELLO -- identifier

   A.B.C  --database object name
   ```

   A unquoted string is similar to an Ada enumeration literal in this respect. Unlike an enumeration literal, however, the list of legal unquoted strings is dynamically extendable to include any arbitrary identifier or database name.

(c) Expressions. MCL expressions are a subset of Ada expressions, with extensions for dealing with unquoted strings. See Appendix I for MCL expression syntax. Note that a carriage return cannot appear between an operand and an operator of an expression, since a single operand it itself a valid expression.

Examples of expressions:

```
HELLO                   --unquoted string

HELLO & GOODBYE         --HELLOGOODBYE

%Q OR (%V AND FALSE)    --variable names begin with a '%'

2 * 3. + 5              --11
```

OPERANDS: An operand of an expression may be a literal, a function call or a variable.

A literal denotes an explicit value of any legal MCL type. Examples of literals:

```
4               --integer literal

4.0             --real literal

TRUE            --Boolean literal

"OFF"           --string literal

OFF             --unquoted string
```

A function call returns a value which can be used as an operand. A function may have its I/O redirected via -|, -< or ->

An MCL variable name always begins with a '%':

```
variable ::=  %identifier[(simple_expression)]
```

A variable takes on a value via the assignment statement.

TABLE 3-3 shows the set of CP variables that are defined and initialized at CP startup time. Some of these control CP options; these options may be modified by assigning a new value to the variable. Others are used as convenient placeholders for strings generated during job invocation.

Table 3-3 -- Pre-defined CP Variables

NAME	TYPE	PURPOSE	INITIAL VALUE
%PROMPT	STRING	Defines the CP user prompt.	":"
%FJOBS	STRING	The names of all jobs within the last job command executed in the foreground.	""
%EXECUTION_TIME	REAL	The execution time of the last completed foreground job command.	0.
%EXIT_STATUS	STRING	The exit status of the last completed foreground job command.	""
%LAST_TASK	STRING	The name of the last background task.	""
%INFORM_DEFAULT_OUT	BOOLEAN	If TRUE, the user is " " informed of each default OUT parameter variable generated.	FALSE
%AUTO_REDEFINE	BOOLEAN	IF TRUE, a default OUT parameter will replace an already existing variable with the same name.	FALSE
%ACTIVE_JOBS	STRING	A list of all currently executing jobs.	""

The MCL user does not explicitly declare variables. Rather, a variable is implicitly declared by its first use. Its value is a string, with implicit conversions performed as required. The pre-defined command LIST_VARS prints to standard output a list of all CP variables and their current values, separated by blanks. This is useful for determining the current state of all CP variables.

```
list_vars_statement ::=  LIST_VARS
```

A variable's value is made up of one or more lexical elements. The user may refer to a lexical element within a variable's current value by means of the substring operator. This operator is encoded

as a parenthesized expression immediately following the variable name (similar to an Ada indexed component [G-3 4.1.1]). The expression indicates the requested substring's lexical position. The result of the substring operation is a string literal.

If the substring operator's value is greater than the number of lexical units in the variable, the operator's value is the empty string "". For example, if

```
: %VAR :=  "this is a character literal"
```

then %VAR(5) refers to the substring "literal".

The fundamental function LEXICAL_LENGTH returns the lexical length of a string. This is useful for loops, as in

```
: -- loop to list and delete all objects in a library
: %DIR := MYLIB'DIRECTORY  -- list of all object in MYLIB
: FOR %I IN 1..LEXICAL_LENGTH(%DIR) LOOP
1/ LIST %DIR(%I)  -- successive words within the variable %DIR
2/ DELETE %DIR(%I)
3/ END LOOP
```

OPERATORS: MCL operators, a subset of Ada, are divided into five classes. They are given below in order of increasing precedence.

```
LOGICAL_OPERATOR ::= AND | OR

RELATIONAL_OPERATOR ::=  = | /= | < | <= | > | >=

ADDING_OPERATOR ::= + | - | &

UNARY_OPERATOR ::= + | - | NOT

MULTIPLYING_OPERATOR ::= * | /
```

Expression evaluation follows the same rules as Ada [G-3].

Expressions can be surrounded by parentheses to impose a specific order on operators. Table 3-4 Summarizes MCL operators.

Table 3-4: MCL Operators

Operator	Operation	Operand Type	Result Type
(Logical)			
AND	conjunction	BOOLEAN	BOOLEAN
OR	inclusive disjunction	BOOLEAN	BOOLEAN
(Relational)			
= /=	equality and inequality	Any MCL Type	BOOLEAN
< <= > >=	test for ordering	Any MCL type	BOOLEAN
(Equality for reals is determined as in Ada [G-3 4.5.8])			
(Adding)			
+	addition	Integer or real	same type
-	subtraction	Integer or real	same type
&	catenation	String or unquoted string	string
(Unary)			
+	identity	Integer or real	same type
-	negation	Integer or real	same type
NOT	logical negation	BOOLEAN	BOOLEAN
(Multiplying)			
*	multiplication	Integer or real	same type
/	division	Integer or real	same type

The membership tests IN and NOT IN test whether a value is within a specified range. The value and the range bounds can be any MCL type but must be of the same type. These operators return a value of type BOOLEAN.

Examples:

```
%N NOT IN 1..10
```

"HELLO" IN %A..%B

TEST1 IN %A..TESTY

TYPE CONVERSIONS. Any unquoted strings within an expression are implicitly converted to strings. In the following discussion, the term STRING refers to both strings and implicitly converted unquoted strings.

Each of the operations described expects its operands to be of specific types. If an operand is not of the proper type, the CP attempts to implicitly convert it into a value of the proper type. If the conversion fails, an error message is printed. This conversion only occurs between closely related types, as shown in Table 3-5.

Table 3-5: Implicit Type Conversions

Proper Operand Type	Closely Related Types
BOOLEAN	STRING(value must be "TRUE" or "FALSE")
INTEGER	STRING(value must be the image of an integer) REAL (truncated to integer)
REAL	STRING(value must be the image of a real number) INTEGER
STRING	BOOLEAN REAL INTEGER

For operators whose operands can be one of several different types, implicit conversion is attempted in the following order: BOOLEAN, INTEGER, REAL, STRING. This ordering implies that: (1) relational operators and catenation can be applied to operands of any type, since any type can be implicitly converted to a string; and (2) the result of adding, subtracting, multiplying or dividing an integer and a real is an integer. Examples of implicit conversion:

```
FLIGHT & SIM            --"FLIGHTSIM"

TRUE AND "FALSE"        --FALSE

3+"5"                   --8

2.3 + 3                 --5

"HELLO" & 2.0           --"HELLO2.0"
```

DATABASE ATTRIBUTES: In addition to the subset of Ada operators presented above, MCL has an additional operator, used to determine an attribute of a database object.

attribute ::= objname'attribute_name

The object and the attribute name may be specified as arbitrary expressions, including unquoted strings or variables. The result of the operation is a string describing the value of the specified attribute for the specified database object.

Examples:

```
MYLIB'MEMBERS

%X'TERMINATED

%X'%Y
```

3.2.2 Processing

The CP parses a user-typed command into a parse tree (via PARSE), and passes this tree to the TREE_INTERPRET task to be executed. TREE_INTERPRET may:

1. update CP variables (the VARIABLE task);
2. execute a program (JOB);
3. evaluate an expression (the EXPRESSION package);
4. manipulate script parameters (the SUBPROG_SIMULATE package);
5. instantiate other TREE_INTERPRET tasks to execute co-routines (the ALLOCATE_INTERPRETER task); or
6. execute a command in the background (the BACKGROUND task).

Errors in user input are handled by the ERROR package.

3.2.2.1 The CP Driver (DRIVER)

The CP Driver is the main program of the CP. It performs various initializations, then loops to interpret MCL commands. At the conclusion of command processing, various postprocessing is performed.

(a) Specification.

```
with GLOBALS, VARIABLE, BACKGROUND, ALLOCATE_INTERPRETER,
     INPUT_STREAM, PARSE, ERROR
procedure DRIVER;
```

(b) Dependencies.

VARIABLE, BACKGROUND, ALLOCATE_INTERPRETER -These global tasks are elaborated in DRIVER's declarative part.

GLOBALS - This package contains types used to refer to an active TREE_INTERPRET task.

INPUT_STREAM -Entries in this package are invoked to initialize the CP's command input stream.

PARSE -This procedure is invoked to parse a user-typed MCL command.

ERROR -Entries in this package are invoked to handle errors.

(c) Algorithm. DRIVER performs the following operations:

1. Since several commands may be interpreted simultaneously (for example co-routines and background commands), global CP data must be managed by CP-wide tasks. These include CP variables (the VARIABLE task), TREE_INTERPRET tasks (the ALLOCATE_INTERPRETER task) and background task names (the BACKGROUND task). These tasks are therefore activated during the elaboration of DRIVER's declaration part.

2. If the CP invocation is, in fact, a resumption of a previously suspended CP session, the CP's context object has an attribute VARIABLE_VALUES, which describes the values of CP variables when the CP was suspended (see the SUSPEND_EXCEPTION below). If this attribute is present, DRIVER invokes the VARIABLE task entry SET to initialize each variable in VARIABLE_VALUES.

3. The CP's parameters are read from the context object. Possible parameters are:

SCRIPT_NAME - The database object from which the CP is to read commands. If supplied, the CP is processing a script.

SCRIPT_PARMS - This string describes IN parameters for any subprogram simulation statements. The SUBPROG_SIMULATE package entry IN_PARAMETERS is invoked to process these SCRIPT parameters.

COMMAND_STRING - a string to be executed by the CP.

4. Any commands contained in startup files are interpreted. These include a system-wide startup file (containing intializations of pre-defined variables), and "CP_STARTUP," which contains user-defined CP startup commands.

5. The CP determines the input stream from which it is to read commands, as follows: If either SCRIPT_NAME or COMMAND_STRING

is supplied, it specifies the command input; otherwise, the CP reads commands from standard input. In any case, INPUT_STREAM.CREATE is invoked to initialize the input stream.

6. The CP activates a TREE_INTERPRET task via ALLOCATE_INTERPRETER.ALLOC

7. DRIVER loops to process MCL commands from the input stream. These commands may raise exceptions, which are caught and handled by DRIVER. The algorithm for this loop and exception handlers can be stated as follows:

```
<<MORE>>

        while not EOF loop

                Invoke BACKGROUND.POLL before reading each
                command to determine if any background tasks
                have completed.

                Invoke PARSE to parse the next command from
                the input stream into a parse tree.

                Pass the parse tree to the TREE_INTERPRET task
                to be interpreted.

        end loop;

        if EOF then
           if operating in interactive mode then goto MORE end if
        end if;

exception

   when COMMAND_ERROR =>

        Flush tokens until newline;
        goto MORE;

   when LOGOFF_EXCEPTION =>

        Perform user and "system-wide shutdown commands;
        Invoke the KAPSE function LOGOFF;

   when SUSPEND_EXCEPTION =>

        Create an attribute VARIABLE_VALUES in the CP's
        context object, which contains a concatenated list
        of variable names and values, and terminate the
        CP session.

   when RETURN_EXCEPTION =>

        Perform user and system-wide shutdown commands;
        Terminate the CP session;
```

3.2.2.2 Input Stream Package (INPUT_STREAM)

The CP reads and parses commands from an input stream.

This input stream may either be an open file or a string. The Input Stream Package contains an entry to create an input stream.

(a) Data Structures. An input stream is described by a variant record, STREAM, whose field describes either an open file or a string.

(b) Specification.

```
with GLOBALS package INPUT_STREAM is

        type STREAM_STRING is access STRING;
        type STREAM_TYPE is ( FROM_STRING, FROM_FILE );
        type STREAM_(FROM: STREAM_TYPE) is
          record
                case FROM is
                   when FROM_STRING =>  STR: STRING_PTR
                   when FROM_FILE =>    FILE: TEXT_IO.IN_FILE;
                end case;
          end record;
        procedure CREATE( FILE_OR_STRING: STREAM_TYPE;
                          FILE: TEXT_IO.IN_FILE;
                          STR: STRING_PTR
                          NEW_STREAM: in out STREAM);
 end INPUT_STREAM;
```

(c) Dependencies. GLOBALS - contains STRING_PTR's declaration.

(d) Entries.

CREATE_STREAM. This entry initializes a STREAM record. If the stream is described by a string, the string is concatenated with an end-of-stream marker.

3.2.2.3 Lexical Analyzer

The lexical analyzer reads lexemes from a specified input stream. Its algorithm is described in [I-2, 3.2.2.2].

(a) Specification.

```
with INPUT_STREAM, GLOBALS

        procedure LEXICAL_ANALYZER( INSTREAM: INPUT_STREAM;

                                    TOKEN: out STRING_PTR);
```

(b) Dependencies.

INPUT_STREAM -Data structures in this package define the type of an input stream.

GLOBALS - contains STRING_PTR's declaration.

(c) Algorithm. The lexical analyzer reads and returns the next lexeme from INSTREAM.

3.2.2.4 The MCL Parser (PARSE)

The MCL Parser parses a command from an input stream.

(a) Specification.

```
with LEXICAL_ANALYZER, ERROR, INPUT_STREAM
        procedure PARSE(S: in STREAM;
                        T: out P_TREE);
```

(b) Dependencies.

INPUT_STREAM -Data structures in this package define the type of an input stream.

LEXICAL_ANALYZER -This procedure is invoked to read lexemes from an input stream.

ERROR -Entries in this package are invoked to handle errors.

(c) Data Structures. A command is parsed via recursive descent into a Diana-like parse tree allocated from the heap. This tree may describe an MCL statement, as well as any notation to be applied over the statement. This latter category includes: REDIRECT_INPUT(-<), REDIRECT_OUTPUT(->), PIPE(-|) and BACKGROUND(-&).

(d) Algorithm. PARSE takes as its IN parameter the input stream from which the command is to be read. It reads tokens from the stream (via the Lexical Analyzer), and forms a parse tree; its goal symbol is a complete MCL statement.

PARSE also contains an exception handler for the interrupt exception, which causes the parse tree to that point to be ignored.

3.2.2.5 The Parse-Tree Interpreter (TREE INTERPRET)

The Parse-Tree Interpreter is the heart of the CP. It is activated to process user-typed commands, to execute a command in the background, and to execute commands as coroutines.

(a) Specification.

```
with JOB, VARIABLE, ERROR, ALLOCATE_INTERPRETER, GLOBALS,
     PARSE, SUBPROG_SIMULATE, EXPRESSION, BACKGROUND
  task TREE_INTERPRET is
       entry INITIALIZE( CMD_STANDARD_INPUT: TEXT_IO.IN_FILE;
                         CMD_STANDARD_OUTPUT: TEXT_IO.OUT_FILE;
                         CONTEXT_CO: STRING_PTR);
       entry NEXT_TREE(TREE : P_TREE);
   end TREE_INTERPRET;
```

(b) Dependencies.

ERROR -Entries in this package are invoked to handle errors.

VARIABLE -Entries in this task are invoked to manipulate CP variables.

JOB -This procedure is called to invoke a program.

BACKGROUND -Entries in this package are invoked to interpret a command in the background.

ALLOCATE_INTERPRETER -Entries in this package are invoked to activate and deactivate TREE_INTERPRET tasks for co-routines.

GLOBALS -This package contains data structures used to refer to an active TREE_INTERPRET task.

PARSE -This procedure is invoked to parse the string argument of an EXEC statement.

SUBPROG_SIMULATE -Entries in this package are invoked to read and write parameters from the CP's context object.

EXPRESSION -Entries in this package are invoked to evaluate an expression.

(c) Data Structures. TREE_INTERPRET maintains two variables, COMMAND_STANDARD_INPUT and COMMAND_STANDARD_OUTPUT, which correspond to the standard input and output for any command it interprets. An executing TREE_INTERPRET task may modify these values without affecting the standard input or output of commands interpreted by any other TREE_INTERPRET task. TREE_INTERPRET also maintains a composite object located in the CP's context object, within which it creates context objects for any programs it invokes.

(d) Entries. INITIALIZE. This entry initializes the standard input and output for any command executed by TREE_INTERPRET. Additionally, a composite object, whose name is supplied as the IN parameter CONTEXT_CO, is created.

NEXT_TREE. This entry serves as a rendezvous point where TREE_INTERPRET receives the next parse tree to be interpreted.

(e) Algorithm. As each parse tree is received, it is passed to a procedure INTERPRET within the task body. INTERPRET performs various actions depending on the command it is to effect, as follows:

PROGRAM_OR_SCRIPT_CALL: INTERPRET generates a unique context object name within CONTEXT_CO. This name is passed to the JOB procedure, which creates the context object and executes the program or interprets the CP script.

ASSIGNMENT: The assignment is performed.

RETURN, SUSPEND or LOGOFF: The POLL entry of the BACKGROUND task is invoked to determine if any background tasks are active, in which case the user is informed and the command is ignored. Otherwise, an appropriate condition is raised to be caught by DRIVER.

SET_INPUT, SET_OUTPUT: The appropriate internal variable (COMMAND_STANDARD_INPUT or COMMAND_STANDARD_OUTPUT) is reset to indicate the standard input or output of any subsequent commands interpreted by TREE_INTERPRET.

ABORT: The B_ABORT entry of the BACKGROUND task is invoked to abort the specified background task.

WAIT: The WAIT entry of the BACKGROUND task is invoked to wait for the completion of the specified background task.

EXPRESSION: The value of the expression is written to COMMAND_STANDARD_OUTPUT.

EXEC: The expression argument is evaluated into a string, which is converted into an input stream via INPUT_STREAM.CREATE. The resulting STREAM is parsed (by PARSE) into a parse tree. INTERPRET then recurses to interpret that parse tree.

IF: Successive conditions are evaluated until one evaluates to TRUE. INTERPRET recurses to execute the subtree associated with that condition. If none of the conditions evaluate to TRUE, no subtree is executed.

LOOP: The iteration clause is evaluated and, if it is TRUE, INTERPRET recurses to execute the loop body. This process is repeated until the iteration clause evaluates to FALSE, or until a command within the loop terminates the loop (EXIT), or the CP session (SUSPEND, LOGOFF, RETURN).

EXIT: Execution of the loop is terminated.

SUBPROGRAM_SIMULATION: Parameter values are initialized via the GET_PARAMETERS entry of the SUBPROG_SIMULATE package. INTERPRET recurses to execute the associated block of commands, then invokes SUBPROG_SIMULATE.PUT_PARAMETERS to write the values of updated OUT parameters back to the context object.

BLOCK: INTERPRET recurses to perform the commands within the block.

REDIRECT_INPUT (-<), REDIRECT_OUTPUT (->): INTERPRET opens (for input) or creates (for output) the specified database object, resets the appropriate internal variable (COMMAND_STANDARD_INPUT or COMMAND_STANDARD_OUTPUT) to indicate that standard input (output) for commands should be the database object, and recurses to interpret commands. When all commands have been processed, the database object is closed, and standard input(output) for commands reverts to its previous value.

PIPE: The ALLOC entry of the ALLOCATE_INTERPRETER task is invoked to initiate a TREE_INTERPRET task for each command within a pipeline. Each TREE_INTERPRET task is passed a subtree describing the command within the pipeline it is to execute. INTERPRET then waits for all these TREE_INTERPRET tasks to complete, at which point ALLOCATE_INTERPRETER.DEALLOC is invoked to deallocate them.

BACKGROUND (-&): The parse tree is passed to the BACKGROUND task to be interpreted.

3.2.2.6 The Globals Package (GLOBALS)

This package contains an access type to the TREE_INTERPRET task, which serves as a means of referring to it in various CP modules.

(a) Specification.

```
with TREE_INTERPRET
  package GLOBALS is
        type STRING_PTR is access STRING;
        type INTERPRETER is limited private;
    private
        type INTERPRETER is access TREE_INTERPRET;
  end GLOBALS;
```

3.2.2.7 The Interpreter Allocator Task (ALLOCATE INTERPRETER)

The Interpreter Allocator task maintains a pool of TREE_INTERPRET tasks.

(a) Specification.

```
with TREE_INTERPRET, GLOBALS
  task ALLOCATE_INTERPRETER is
       entry ALLOC (ACTIVATED_INTERPRETER: out INTERPRETER);
       entry DEALLOC(ACTIVATED_INTERPRETER: INTERPRETER);
end ALLOCATE_INTERPRETER;
```

(b) Dependencies. TREE_INTERPRET -ALLOCATE_INTERPRETER maintains a pool of these tasks.

GLOBALS -This package contains data structures used to refer to an active TREE_INTERPRET task.

(c) Data Structures.

The ALLOCATE_INTERPRETER task maintains an array of access variables to TREE_INTERPRET tasks.

(d) Entries.

ALLOC. This entry activates a TREE_INTERPRET task from the pool.

DEALLOC. This entry aborts the specified TREE_INTERPRET task and returns it to the pool, from whch it may later be reactivated.

3.2.2.8 The Background Task Manager (BACKGROUND)

A background command is processed by activating a TREE_INTERPRET task to interpret it. The Background Task Manager controls the execution of background commands, and associates a unique name with each background task.

(a) Specification.

```
with ALLOCATE_INTERPRETER, GLOBALS, ERROR

  task BACKGROUND is

  entry START(COMMAND: P_TREE;

              COMMAND_STANDARD_INPUT: TEXT_IO.IN_FILE;

              COMMAND_STANDARD_OUTPUT: TEXT_IO.OUT_FILE;

              TASK_ID: out STRING_PTR);

  entry WAIT( TASK_ID: STRING_PTR);

  entry B_ABORT (TASK_ID: STRING_PTR);

  entry POLL(ACTIVE_TASKS: out STRING_STR);

end BACKGROUND;
```

(b) Dependencies.

ALLOCATE_INTERPRETER -Entries in this package are invoked to activate and deactivate a TREE_INTERPRET task.

GLOBALS - This package contains data structures used to refer to an active TREE_INTERPRET task.

ERROR - Entries in this package are invoked to handle errors.

(c) Data structures. BACKGROUND maintains a list of all TREE_INTERPRET tasks activated to process background commands, as well as the unique names of these tasks.

(d) Entries.

START. This entry is invoked to start the execution of a command (specified via the IN parameter COMMAND) in the background. ALLOCATE_INTERPRETER.ALLOC is invoked to activate a new TREE_INTERPRET task. START generates a unique name for the task, and passes this name, along with COMMAND_STANDARD_INPUT and COMMAND_STANDARD_OUTPUT to the task's INITIALIZE entry. The task is passed COMMAND via its NEXT_TREE entry, and the user is informed of the task name.

WAIT. This entry causes the CP to wait for the termination of the specified background task before accepting subsequent user commands.

B_ABORT. This entry aborts the background TREE_INTERPRET task identified by TASK_ID.

POLL. This entry polls the status of all currently active background tasks via conditional entry calls [G-3, 9.7.2]. If any task is done (i.e. is ready to receive its next command), it is deactivated via ALLOCATE_INTERPRETER.DEALLOC, and the user is informed that the background task completed.

POLL returns as it OUT parameter a concatenated list of names of all currently active background tasks.

3.2.2.9 Job Invocation (JOB)

Some MCL commands are executed via internal CP actions, while others are effected by invoking a program. This latter category includes a call on a linked executable progra, as well as a request to interpret a CP script (which requires the implicit invocation of a CP to interpret it). A request for program invocation is referred to as a job. The JOB procedure effects job invocation.

(a) Specification.

```
with ERROR, VARIABLE, GLOBALS

        procedure JOB( CONTEXT_NAME: STRING_PTR;

                       TJOB: P_TREE);
```

(b) Dependencies.

ERROR - Entries in this package are invoked to handle errors.

GLOBALS - contains STRING_PTR's declaration.

VARIABLE - Entries in this task are invoked to manipulate CP variables.

(c) Algorithm. JOB takes as its IN parameters a parse tree that describes a job, and the name of the context object it is to create for that job. JOB must:

1. create the specified context object;

2. write IN parameter values to the context object;

3. invoke the appropriate program;

4. wait for the program to complete; and

5. update OUT parameters modified during the program's execution.

The IN parameter values to be written are contained in the parse tree. The name of the program or script to be executed is expanded (via the PROGRAM_SEARCH_LIST attribute of the CP's context object) into a full database object name. If the category of that database object indicates that it is a program, the program is invoked; if the category indicates it is a text file, the CP is invoked to process the script as its command input. EXECUTE waits for the program's completion via the KAPSE call AWAIT_PROGRAM, then updates OUT parameters as follows:

The KAPSE call GET_ATTRIBUTE is invoked to return a parameter string. For each parameter in the string, the parse tree is examined to determine whether the parameter was specified by the user as part of program invocation, either by position or by name. If so, the parameter's actual name is, in fact, the name of a variable, which is updated to contain the new OUT parameter value.

If the parameter was not specified by the user, a "default OUT parameter" variable is generated. The value of this variable is set to the OUT parameter's value.

If the parameter was supplied by the user but was not a variable , a warning is issued, and a default OUT parameter is generated, as above. The program's exit status is computed, and, if it is "OK", all database objects within the context object are deleted.

3.2.2.10 Subprogram Simulation Package (SUBPROG_SIMULATE)

When the CP executes a subprogram simulation statement, it must perform preprocessing to import parameters from its context object, and postprocessing, to return updated OUT parameter values to the context object.

It does so by invoking SUBPROG_SIMULATE package entries.

(a) Data Structures. The SUBPROG_SIMULATE package maintains a private variable PARM_STRING, which contains IN parameter values as read by DRIVER.

(b) Specification.

```
with ERROR, VARIABLE, GLOBALS
        package SUBPROG_SIMULATE is
           procedure IN_PARAMETERS(PARM_STRING: STRING_PTR);
           procedure GET_PARAMETERS(SPEC: P_TREE);
           procedure PUT_PARAMETERS(SPEC: P_TREE);
         private
           PARM_STRING: STRING_PTR;
        end SUBPROG_SIMULATE;
```

(c) Dependencies.

ERROR - ERROR package entries are invoked to handle errors.

VARIABLE - VARIABLE task entries are invoked to manipulate CP variables.

GLOBALS -contains STRING_PTR's declaration.

(d) Entries.

IN_PARAMETERS. This entry is invoked by the DRIVER to store the IN parameter string (as read from the CP's context object) in the private variable PARM_STRING.

GET_PARAMETERS. This entry is invoked to initialize IN parameter values. It takes as its IN parameter a parse tree describing a subprogram simulation statement.

The algorithm for GET_PARAMETERS is as follows:

For each parameter association in the parse tree, a value is obtained from PARM_STRING (or from the parse tree, if a default value was specified), and stored in a CP variable whose name corresponds to the formal parameter name.

PUT_PARAMETERS: This entry is invoked at the conclusion of script processing, to write OUT parameter values to the context object. The parse tree is scanned to build a new parameter string containing updated OUT parameter values. If the script describes a function, its return value is also placed in the string, which is then written via the KAPSE call SET_ATTRIBUTE. Finally, any CP variables defined in GET_PARAMETERS are deleted.

3.2.2.11 Variable Task (VARIABLE)

The VARIABLE Task is responsible for maintaining CP variables. It contains entries for defining a CP variable, assigning a value to a CP variable, and fetching the current value of a CP variable.

(a) Data Structures. A variable is represented by a record allocated from the heap containing the field VALUE, which describes the value of the CP variable.

(b) Specification.

```
with ERROR, GLOBALS

task VARIABLE is

   entry DEFINE (NAME:STRING_PTR);

   entry SET (NAME:STRING_PTR;VALUE:STRING_PTR);

   entry FETCH(NAME:STRING_PTR; VALUE: out STRING_PTR);

end VARIABLE;
```

(c) Dependencies.

ERROR-ERROR package entries are invoked to handle errors.
GLOBALS - contains STRING_PTR's declaration.

(d) Entries. DEFINE, FETCH and SET. These entries are invoked to define a new CP variable, set its value and fetch that value. If a referenced variable does not exist, an implicit declaration is generated.

3.2.2.12 String Conversion Package (STRING_CONVERT)

During expression evaluation, it may be necessary to convert the string value of an MCL variable into some other type required to perform a particular operation. In these cases, the STRING_CONVERT package is called upon to perform the necessary conversion.

(a) Specification.

```
with GLOBALS
package STRING_CONVERT is
procedure TO_INTEGER (STR:STRING_PTR; VALUE:OUT INTEGER)
procedure TO_REAL (STR:STRING_PTR;VALUE:OUT REAL)
procedure TO_BOOLEAN (STR:STRING_PTR;VALUE:OUT BOOLEAN)
end STRING_CONVERT;
```

(b) Dependencies.

GLOBALS - Contains STRING_PTR's declaration.

(c) Entries.

TO_INTEGER, TO_REAL and TO_BOOLEAN: These entries are invoked to convert a string value into a value of type INTEGER, BOOLEAN or REAL. Each takes as its IN parameter the string to be converted. Conversion is performed via the Ada VALUE attribute. If the conversion is successful (i.e., the DATA_ERROR exception is not raised), the converted value of the requested type is returned as an OUT parameter. Otherwise, an error condition is raised.

3.2.2.13 Operator Package (OPERATOR)

During expression evaluation, it may be necessary to perform a pre-defined operation (such as addition or subtraction) on one or more operands. The operator package performs such operations.

(a) Specification

```
with GLOBALS, STRING_CONVERT
package OPERATOR is
   function EVALUATE  (OPERATOR:STRING_PTR;
                       STRING1:STRING_PTR;
                       STRING2:STRING_PTR;
                       VALUE:OUT STRING_PTR);

end OPERATOR;
```

(b) Dependencies.

STRING_CONVERT -Entries in this package are invoked to convert a string into some other MCL type.

GLOBALS - contains STRING_DTR's declaration.

(c) Entries.

EVALUATE. This entry is invoked to perform an arbitrary, pre-defined operation on a group of strings. It takes as its IN parameters the string argument(s) and an operation to be performed on the string(s). It performs the operation, and returns as its OUT parameter a string representing the result of the operation.

3.2.2.14 Expression Package (EXPRESSION)

The Expression Package is responsible for evaluating expressions.

(a) Specification.

```
with ERROR, VARIABLE, JOB, OPERATOR, GLOBALS
package EXPRESSION is
  procedure EVALUATE(TREE: P_TREE;

                     VALUE: out STRING_PTR);

end EXPRESSION;
```

(b) Dependencies.

ERROR -ERROR package entries are invoked to handle errors.

VARIABLE -Entries in this task are invoked to fetch the value of CP variables.

JOB -This procedure is invoked to effect a function call.

OPERATOR -Entries in this package are invoked to evaluate operators encountered during expression evaluation.

GLOBALS -Contains STRING_PTR's declaration.

(c) Entries.

EVALUATE. This entry is invoked to evaluate an arbitrary expression. It receives as its IN parameter a parse tree which describes an expression, and returns as it OUT parameter a string describing the expression's value.

3.2.2.15 Error Package (ERROR)

The ERROR package contains facilities for printing error messages and raising the COMMAND_ERROR exception (to be caught by DRIVER).

(a) Data Structures. The ERROR package contains an enumeration type named ERRORS that consists of an enumeration literal for each CP error. Associated with ERRORS is a composite object containing explanatory text to be printed for each error.

(b) Specification.

```
with GLOBALS
PACKAGE ERROR is

type ERRORS is -- a literal for each error type

    (TERMINATOR_EXPECTED, NO_SUCH_PROGRAM, . . .);

procedure MESSAGE (ERROR_NAME:ERRORS; MESSAGE_STRING:STRING_PTR);

end ERROR;
```

(c) Dependencies.

GLOBALS - contains STRING_PTR's declaration.

(d) Entries.

MESSAGE -prints an error message. IN parameters are the error name and an optional message string which should precede the error text. If MESSAGE_STRING is specified, MESSAGE prints it. (This message string may, for example, be the name of the specific user-typed token which caused the error.) MESSAGE then prints the explanatory text associated with the error, and raises the COMMAND_ERROR exception.

4.0 QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Testing of the Command Processor will be performed at three levels:

1) Unit Testing

2) Integration Testing

3) Functional Testing

See the CPDP (TBD) for a detailed discussion of the methodology to be used while constructing the MAPSE. The tests described below will be repeated on both MAPSE Configurations, VM370 and Perkin-Elmer 8132.

4.1.1 Unit Testing

Each subprogram package and task which makes up the CP program will be unit tested. This involves constructing a specific environment around the module to be tested, and then invoking it with specific arguments. The actions that the module takes can then be observed and compared with the expected actions.

Each fundamental program will also be unit tested.

4.1.2 Integration Testing

During integration testing, the CP modules will be integrated one-at-a-time and tested with previous tested subsets of the total CP program. Additionally, fundamental programs will be integrated with the CP program.

4.1.3 Functional Testing

A set of MCL commands will be developed which will verify that the CP requirements contained in Section 3 of this Specification are fully operational. The tests will be utilized interactively and in script mode. The test results and the source code for the CP and fundamental programs will be available for inspection. In addition, portions of the test sequence may be exercised for demonstration purposes.

PAGE LEFT BLANK INTENTIONALLY

10.0 APPENDIX

10.1 BNF For MAPSE Command Language (MCL)

--LEXICAL ELEMENTS

```
upper_case_letter ::= A | B | C | D | E | F | G | H | I |
                      J | K | L | M | N | O | P | Q | R | S
                      | T | U | V | W | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

special_characters ::= " | # | % | & | ' | ( | ) | star | + |
                       comma | - | dot | / | colon | semicolon |
                       < | = | > | underscore | parallel_bar

star ::= <*>

comma ::= <,>

dot ::= <.>

colon ::= <:>

semicolon ::= <;>

underscore ::= <_>

parallel_bar ::= <|>

blank ::=  < >
```

```
lower_case_letter ::= a | b | c | d | e | f | g | h | i | j
                    | k | l | m | n | o | p | q | r | s | t | u
                    | v | w | x | y | z

other_special_characters ::=
 ! | $ | question_mark | @ | [ | / | ] | ^ | '| { | } | ~

question_mark ::= <?>

identifier ::= letter{[underscore]letter_or_digit}
 -- identical to ADA

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

numeric_literal ::= decimal_number

decimal_number ::= integer [.integer][exponent]

integer ::= digit{[underscore]digit}

exponent ::= E [+] integer | E - integer

character_string ::= "{character}"

character ::= letter_or_digit | special_characters | blank |
       other_special_characters

database_literal ::= partition | objname

partition ::= [.]id_or_av_list_or_star{.id_or_av_list_or_star}

objname ::= [.]id_or_av_list{.id_or_av_list}

id_or_av_list_or_star ::= identifier | attribute_value_list|

id_or_av_list ::= identifier | attribute_value_list

attribute_value_list ::=

 (attribute_association {comma attribute_association})

attribute_association ::= [attribute_name =>] expression

attribute_name ::= identifier
```

```
-- EXPRESSIONS

 -- Subset of ADA expressions


expression ::= limited_expression | unquoted string
 -- A single unquoted string can be an expression,
 -- except within an expression statement.


unquoted string ::= database_literal
 -- Includes simple identifiers.


limited_expression ::= relation { logical_operator relation}



relation ::= simple_expression
     |  simple_expression_or_unquoted string  relational_operator
        simple_expression_or_unquoted string
     |  simple_expression_or_unquoted string [NOT] IN discrete_range



simple_expression ::=  [unary_operator]term
      |  [unary_operator]term_or_unquoted string  adding_operator
         term_or_unquoted string
         { adding_operator term_or_unquoted string}



term ::= primary
      |  primary_or_unquoted string  multiplying_operator
         primary_or_unquoted string
         { multiplying_operator term_or_unquoted string}
      | attribute



primary ::= literal | variable |
     (expression) | function_call



simple_expression_or_unquoted_string ::=
 simple_expression | unquoted_string
```

```
term_or_unquoted_string ::=
 term | unquoted_string


primary_or_unquoted_string ::=
 primary | unquoted_string


logical_operator ::= AND | OR


relational_operator ::= = | /= | < | <= | > | >=


unary_operator ::= + | - | NOT


adding_operator ::= + | - | &


multiplying_operator ::= * | /




literal ::= character_string | numeric_literal




variable ::= %identifier[(expression)]
 -- A variable name must begin with a '%'.
 -- The variable name is optionally followed by an substring
    index.


attribute ::=
  primary_or_unquoted_string'primary_or_unquoted_string
 -- Evaluates to the value of the attribute associated
 -- with the database object.
```

```
--------------------------------------------------------------------

        --COMMANDS

command_input ::= command {terminator command}

terminator ::= newline | semicolon


newline ::= [comment] <CR>

comment ::= -- { character }
 -- As in ADA

command ::=
[label] statement{-| statement } [-< objname ] [ -> objname ] [ -& ]
 -- The notation -| between two commands indicates that
 -- the command are to be connected via a pipe.
 -- Standard output of the left side of a pipe becomes
 -- standard input of the right side of a pipe.
 -- The notation -& indicates that the command's execution should
 -- take place in the background. A task is generated to perform
 -- the command.
 -- The notation -< is used to indicate that the leftmost command's
 -- standard input should be directed from OBJNAME.
 -- The notation -> is used to indicate that the rightmost command's
 -- standard output should be directed to OBJNAME.

label ::= <<identifier>>

statement ::= simple_statement | compound_statement



simple_statement ::= null_statement |
              program_or_script_call_statement |
              assignment_statement |
              return_statement  |
              suspend_statement  |
              logoff_statement  |
              set_input_statement  |
              set_output_statement |
              abort_statement |
              wait_statement |
              list_vars_statement|
              exit_statement |
              expression_statement |
              exec_statement |
```

```
compound_statement ::= if_statement | loop_statement
                     | subprogram_simulation_statement
                     | block_statement


null_statement ::= <>



program_or_script_call_statement ::= objname[actual_parameter_part]
 -- Similar to ADA procedure invocation.
 -- OBJNAME may be a linked executable object or a CP script.
 -- Invocation syntax is the same for either.


actual_parameter_part ::= [(] parameter_association
                          {separator parameter_association}[)]


separator ::= blank | comma
 -- Parameters are separated by blank or comma.
 -- However, if the user wishes to continue the parameter list
 -- on the next line, a comma is required.


parameter_association ::= [ formal_parameter => ] actual_parameter
 -- Named and positional parameters can be in any order in
 -- the parameter association. For example,
 --      prog  1  parm4 => 2  5
 -- The semantics of this ordering is as follows:
 -- All positional parameters are grouped together, followed
 -- by all named parameters.
 --
 -- Parameter values can be specified more than once in the
 -- same command invocation. For example,
 --      prog 1 parm4 => 2 5 parm4 => 3
 -- The CP also implements default OUT parameters, as follows:
 -- If, at the end of parameter specification, an OUT
 -- parameter has been omitted, the CP will implicitely
 -- declare a variable named %formal_parameter_name, and will
 -- generate a parameter specification of the form
 --      formal_parmameter => %formal_parameter_name
```

```
formal_parameter ::= identifier


actual_parameter ::= expression | help_mark


help_mark ::= question_mark | colon
 -- Instead of typing an actual parameter value, the user can type
 -- a question mark to find out information about the parameter.
 -- Action taken depends on the "help" attributes of the
 -- invoked program or script.
 -- If those attributes indicate that the program
 -- wishes to handle help requests itself, the
 -- actual_parameter_part is considered to be complete, and the
 -- program is invoked.  The program will presumably take special
 -- actions when it encounters the '?' or ':' as an actual parameter
 -- value.
 -- If the "help" attributes indicate that help files are
 -- available, the CP prints the appropriate
 -- file, and allows the user to continue specifying the
 -- actual_parameter_part for the program or script.


function_call ::= objname([actual_parameter_part])
 -- Unlike a program call, the function parameters
 -- must be surrounded by parentheses.
 -- As in a program call, help may be requested
 -- on a per-parameter basis.


assignment_statement ::= attribute := expression  |
                         variable := expression
      -- As in ADA, except that attribute references to database
      -- objects can appear on the statement's left hand side.



return_statement ::= RETURN [ expression ]
 -- Causes termination of the CP. Control is returned to the
 -- CP's invoker.
 -- If the CP was processing a script which described a
 -- function, the function is given the return value described
 -- by the expression.
```

```
suspend_statement ::= SUSPEND objname
 -- Suspend the current CP's execution and return control to
 -- the CP's invoker.
 -- The CP's context object is saved in OBJNAME, which may be
 -- used as RESUME's argument.


logoff_statement ::= LOGOFF
 -- The current CP session is terminated, along with the
 -- entire MAPSE session.


set_input_statement ::= SET_INPUT [objname]
set_output_statement ::= SET_OUTPUT [objname]
 -- Resets standard input or output for all subsequent commands.


abort_statement ::= ABORT tasknames
 -- Abort the specified background tasks.


wait_statement ::= WAIT tasknames
 -- Wait for the specified background tasks to complete before
 -- accepting more commands.


tasknames ::= identifier {separator identifier}


expression_statement ::= limited_expression
 -- Any expression is legal but a single unquoted string.


exec_statement ::= EXEC expression
 -- Executes the string described by the expression as a command.


if_statement ::= IF condition THEN command_input
             { ELSIF condition THEN command_input}
             [ ELSE command_input ] END IF
 -- Same as ADA


condition ::= expression
 -- Must evaluate to boolean.

loop_statement ::= [iteration_clause]  basic_loop
 -- Loops are legal in interactive as well as script mode.


iteration_clause ::= FOR variable IN discrete_range  |
                     WHILE condition
         -- As in ADA
```

```
discrete_range ::= expression .. expression


basic_loop ::= LOOP command_input END LOOP

exit_statement ::= EXIT

block_statement ::= BEGIN command_input END



subprogram_simulation_statement ::=
        subprogram_specification IS block_statement [identifier]
 -- Execution of this statement causes the CP to import
 -- from the CP's context object
 -- parameters specified in the SUBPROGRAM_SPECIFICATION,
 -- execute the commands contained in BLOCK_STATEMENT,
 -- then write updated OUT parameter values back to
 -- the CP's context object.
 -- This enables the user to create a CP script which simulates
 -- an ADA program.


subprogram_specification ::=
 PROCEDURE identifier [formal_part]      |
 FUNCTION  identifier [formal_part] RETURN subtype_indication
    -- General scheme for script parameters: their formal name
    -- must have the same format as any CP variable - i.e.,
    -- %identifier. Thus, they can be treated as any CP
    -- variable.
    -- The program invoking the CP script may optionally
    -- omit the leading '%' when specifying a script's
    -- formal parameter name.



formal_part ::=
( parameter_declaration { terminator parameter_declaration }  )
 -- Parameter declarations can be separated by newline or semicolon.


parameter_declaration ::=
CP_identifier_list: mode subtype_indication [:= expression]


CP_identifier_list ::= variable { , variable }

mode ::= [IN] | OUT | IN OUT


subtype_indication ::= INTEGER | BOOLEAN | STRING | REAL
 -- Any legal MCL type.

list_vars_statement ::= LIST_VARS
```

10.2 Statements and Fundamental Programs

The following tables summarize those MCL commands which are statements (i.e., executed via internal CP actions) and those which are effected via invocation of a fundamental program.

TABLE 10-1: STATEMENTS

Name	Reason for Internal Execution
assignment list_vars	References internal CP variables.
return logoff suspend	Checks for active internal background tasks, terminate CP invocation.
wait abort	Affects the status of background tasks.
set_input set_output	Modifies the standard I/O files for any invoked from the CP.
if loop exit	Modifies flow of MCL command execution.
subprogram_simulation	Reads and writes parameters from CP's context object.
block	Notation following the block (I/O redirection, pipe or background) must be interpreted by the CP.
exec	String is evaluated by the CP as if it were typed by the user as a command

TABLE 10-2: FUNDAMENTAL PROGRAMS

Name	Comments
GET	Values returned as OUT parameters.
PUT	Values received as IN parameters.
HELP	
RESUME	Invokes the CP. RESUME's argument specifies the CP's context object.
STOP	KAPSE function.
START	"
CANCEL	"
STATUS	"
database manipulation commands	"
LEXICAL_LENGTH	

10.3 Ada - MCL Comparison

TABLE 10-3: COMMANDS

MCL Command	Ada Statement	Differences
Program Call	Procedure Call	1. program name replaces the procedure name. 2. parentheses surrounding the actual parameter part may be omitted. 3. parameter associations may be separated by a blank. 4. positional and named parameters may be freely mixed. 5. named parameters may be repeated. 6. default OUT parameters are generated. 7. parameter help available.
assignment	assignment	left-hand side may describe a database attribute.
get	get	1. reads from standard input only. 2. reads text only. 3. reads a variable number of values.
put	put	1. writes to standard output only. 2. writes text only. 3. writes a variable number or values.

TABLE 10-3: COMMANDS (Cont'd.)

if	if	___
loop	loop	____
exit	exit	1. only the enclosing loop maybe exited. 2. no condition may be assciated with the exit.
return	return	____
set_input set_output	set_input set_output	1. takes as its argument a database object name rather than an open file. 2. if its argument is omitted, standard input(output) reverts to the CP's standard (output).
abort	abort	____
block	block	____
subprogram simulation	subprogram specification	formal parameter names must follow the format of CP variables (i.e., must begin with '%').

TABLE 10-4: LANGUAGE ELEMENTS

MCL element	Ada element	Differences
numeric literals	numeric literals	___
string literals	string literal	may be unquoted.
boolean literals	boolean literals	___
types	types	pre-defined set.
variables	variables	1. name must be preceded by '&'. 2. type is STRING only. 3. Substring operator.
expressions	expressions	no xor, and then, or else, mod, rem or exponentiation operators.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


FILMED
8